# objc ↑↓ Optimizing Collections

By Károly Lőrentey

For more books and articles visit us at http://objc.io
Email: mail@objc.io
Twitter: @objcio

## Preface



## 1 Introduction



## 2 Sorted Arrays



## 3 The Swiftification of NSOrderedSet

## 4 Red-Black Trees



## 5 The Copy-On-Write Optimization



## 6 B-Trees

# Preface

On the surface level, this book is about making fast collection implementations. It presents many different solutions to the same simple problem, explaining each approach in detail and constantly trying to find new ways to push the performance of the next variation even higher than the last.

But secretly, this book is really just a joyful exploration of the many tools Swift gives us for expressing our ideas. This book won't tell you how to ship a great iPhone app; rather it teaches you tools and techniques that will help you become better at expressing your ideas in the form of Swift code.

The book has grown out of notes and source code I made in preparation for a talk I gave at the dotSwift 2017 Conference. I ended up with so much interesting material I couldn't possibly fit into a single talk, so I made a book out of it. (You don't need to see the talk to understand the book, but the video is only 20 minutes or so, and dotSwift has done a great job with editing so that I almost pass for a decent presenter. Also, I'm sure you'll love my charming Hungarian accent!)

# Who Is This Book For?

At face value, this book is for people who want to implement their own collection types, but its contents are useful for anyone who wants to learn more about some of the idiosyncratic language facilities that make Swift special. Getting used to working with algebraic data types, or knowing how to create *swifty* value types with copy-on-write reference-counted storage, will help you become a better programmer for everyday tasks too.

This book assumes you're a somewhat experienced Swift programmer. You don't need to be an expert, though: if you're familiar with the basic syntax of the language and you've written, say, a thousand lines of Swift code, you'll be able to understand most of it just fine. If you need to get up to speed on Swift, I highly recommend another objc.io book, *Advanced Swift* by Chris Eidhof, Ole Begemann, and Airspeed Velocity. It picks up where Apple's *The Swift Programming Language* left off, and it dives deeper into Swift's features, explaining how to use them in an idiomatic (aka *swifty*) way.

Most of the code in this book will work on any platform that can run Swift code. In the couple of cases where I needed to use features that are currently available in neither the standard library nor the cross-platform Foundation framework, I included

platform-specific code supporting Apple platforms and GNU/Linux. The code was tested using the Swift 4 compiler that shipped in Xcode 9.

# New Editions of This Book

From time to time, I may publish new editions of this book to fix bugs, to follow the evolution of the Swift language, or to add additional material. You'll be able to download these updates from the book's product page on Gumroad. You can also go there to download other variants of the book; the edition you're currently reading is available in EPUB, PDF, and Xcode playground formats. (These are free to download as long as you're logged in with the Gumroad account you used to purchase the book.)

# Related Work

I've set up a GitHub repository where you can find the full source code of all the algorithms explained in the text. This code is simply extracted from the book itself, so there's no extra information there, but it's nice to have the source available in a standalone package in case you want to experiment with your own modifications.

You're welcome to use any code from this repository in your own apps, although doing so wouldn't necessarily be a good idea: the code is simplified to fit the book, so it's not quite production quality. Instead, I recommend you take a look at *BTree*, my elaborate ordered collections package for Swift. It includes a production-quality implementation of the most advanced data structure described in this book. It also provides tree-based analogues of the standard library's `Array`, `Set`, and `Dictionary` collections, along with a flexible `BTree` type for lower-level access to the underlying structure.

*Attabench* is my macOS benchmarking app for generating microbenchmark charts. The actual benchmarks from this book are included in the app by default. I highly recommend you check out the app and try repeating my measurements on your own computer. You may even get inspired to experiment with benchmarking your own algorithms.

# Contacting the Author

If you find a mistake, please help me fix it by filing a bug about it in the book's GitHub repository. For other feedback, feel free to contact me on Twitter; my handle is *@lorentey*. If you prefer email, write to *collections@objc.io*.

# How to Read This Book

I'm not breaking new ground here; this book is intended to be read from front to back. The text often refers back to a solution from a previous chapter, assuming the reader has, uhm, done their job. That said, feel free to read the chapters in any order you want. But try not to get upset if it makes less sense that way, OK?

This book contains lots of source code. In the Xcode playground version of the book, almost all of it is editable and your changes are immediately applied. Feel free to experiment with modifying the code — sometimes the best way to understand it is to see what happens when you change it.

For instance, here's a useful extension method on `Sequence` that shuffles its elements into random order. It has a couple of FIXME comments describing problems in the implementation. Try modifying the code to fix these issues!

```
#if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
import Darwin // for arc4random_uniform()
#elseif os(Linux)
import Glibc // for random()
#endif

extension Sequence {
  public func shuffled() -> [Iterator.Element] {
    var contents = Array(self)
    for i in 0 ..< contents.count - 1 {
      #if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
        // FIXME: This breaks if the array has 2^32 elements or more.
        let j = i + Int(arc4random_uniform(UInt32(contents.count - i)))
      #elseif os(Linux)
```

```
            // FIXME: This has modulo bias. Also, `random` should be seeded by calling
            ↪ `srandom`.
            let j = i + random() % (contents.count - i)
        #endif
        contents.swapAt(i, j)
      }
      return contents
    }
}
```

To illustrate what happens when a piece of code is executed, I sometimes show the result of an expression. As an example, let's try running `shuffled` to demonstrate that it returns a new random order every time it's run:

```
▶ (0 ..< 20).shuffled()
  [3, 19, 13, 11, 18, 4, 12, 8, 10, 14, 5, 15, 0, 1, 16, 7, 6, 17, 9, 2]
▶ (0 ..< 20).shuffled()
  [8, 18, 6, 17, 2, 5, 16, 4, 9, 13, 11, 0, 1, 14, 15, 12, 7, 19, 10, 3]
▶ (0 ..< 20).shuffled()
  [9, 1, 6, 8, 17, 3, 13, 7, 16, 4, 18, 14, 5, 0, 15, 2, 19, 12, 10, 11]
```

In the playground version, all this output is generated live, so you'll get a different set of shuffled numbers every time you rerun the page.

# Acknowledgments

This book wouldn't be the same without the wonderful feedback I received from readers of its earlier drafts. I'm especially grateful to *Chris Eidhof*; he spent considerable time reviewing early rough drafts of this book, and he provided detailed feedback that greatly improved the final result.

*Ole Begemann* was the book's technical reviewer; no issue has escaped his meticulous attention. His excellent notes made the code a lot swiftier, and he uncovered amazing details I would've never found on my own.

*Natalye Childress*'s top-notch copy editing turned my clumsy and broken sentences into an actual book written in proper English. Her contribution can't be overstated; she fixed multiple things in almost every paragraph.

Naturally, these nice people are not to be blamed for any issues that remain in the text; I'm solely responsible for those.

I'd be amiss not to mention *Floppy*, my seven-year-old beagle: her ability to patiently listen to me describing various highly technical problems contributed a great deal to their solutions. Good girl!

# Introduction

1

Swift's concept of a collection is one of the core abstractions in the language. The primary collections in the standard library — arrays, sets, and dictionaries — are used in essentially all Swift programs, from the tiniest scripts to the largest apps. The specific ways they work feels familiar to all Swift programmers and gives the language a unique personality.

When we need to design a new general-purpose collection type, it's important we follow the precedent established by the standard library. It's not enough to simply conform to the documented requirements of the `Collection` protocol: we also need to go the extra mile to match behavior with standard collection types. Doing this correctly is the essence of the elusive property of *swiftiness*, which is often so hard to explain, yet whose absence is so painfully noticeable.

# Copy-On-Write Value Semantics

The most important quality expected of a Swift collection is *copy-on-write value semantics*.

In essence, *value semantics* in this context means that each variable holding a value behaves *as if* it held an independent copy of it, so that mutating a value held by one variable will never modify the value of another:

```
var a = [2, 3, 4]
let b = a
a.insert(1, at: 0)
▶ a
  [1, 2, 3, 4]
▶ b
  [2, 3, 4]
```

In order to implement value semantics, the code above needs to copy the underlying array storage at some point to allow the two array instances to have different elements. For simple value types (like `Int` or `CGPoint`), the entire value is stored directly in the variable, and this copying happens automatically every time a new variable is initialized or whenever a new value is assigned to an existing variable.

However, putting an Array value into a new variable (e.g. when we assign to b) does *not* copy its underlying storage: it just creates a new reference to the same heap-allocated buffer, finishing in constant time. The actual copying is deferred until one of the values using the same shared storage is mutated (e.g. in insert). Note though that mutations only need to copy the storage if the underlying storage is shared. If the array value holds the only reference to its storage, then it's safe to modify the storage buffer directly.

When we say that Array implements the *copy-on-write* optimization, we're essentially making a set of promises about the performance of its operations so that they behave as described above.

(Note that full value semantics is normally taken to mean some combination of abstract concepts with scary names like *referential transparency*, *extensionality*, and *definiteness*. Swift's standard collections violate each of these to a certain degree. For example, indices of one Set value aren't necessarily valid in another set, even if both sets contain the exact same elements. Therefore, Swift's Set isn't *entirely* referentially transparent.)

# The SortedSet Protocol

To get started, we first need to decide what task we want to solve. One common data structure currently missing from the standard library is a sorted set type, i.e. a collection like Set, but one that requires its elements to be Comparable rather than Hashable, and that keeps its elements sorted in increasing order. So let's make one of these!

The sorted set problem is such a nice demonstration of the various ways to build data structures that this entire book will be all about it. We're going to create a number of independent solutions, illustrating some interesting Swift coding techniques.

Let's start by drafting a protocol for the API we want to implement. Ideally we'd want to create concrete types that conform to this protocol:

```
public protocol SortedSet: BidirectionalCollection, SetAlgebra {
  associatedtype Element: Comparable
}
```

A sorted set is all about putting elements in a certain order, so it seems reasonable for it to implement BidirectionalCollection, allowing traversal in both front-to-back and back-to-front directions.

SetAlgebra includes all the usual set operations like union(_:), isSuperset(of:), insert(_:), and remove(_:), along with initializers for creating empty sets and sets with particular contents. If our sorted set was intended to be a production-ready implementation, we'd definitely want to implement it. However, to keep this book at a manageable length, we'll only require a small subset of the full SetAlgebra protocol — just the two methods contains and insert, plus the parameterless initializer for creating an empty set:

```
public protocol SortedSet: BidirectionalCollection, CustomStringConvertible,
 ↪  CustomPlaygroundQuickLookable where Element: Comparable {
  init()
  func contains(_ element: Element) -> Bool
  mutating func insert(_ newElement: Element) -> (inserted: Bool, memberAfterInsert:
   ↪  Element)
}
```

In exchange for removing full SetAlgebra conformance, we added CustomStringConvertible and CustomPlaygroundQuickLookable; this is convenient when we want to display the contents of sorted sets in sample code and in playgrounds.

The protocol BidirectionalCollection has about 30 member requirements (things like startIndex, index(after:), map, and lazy). Thankfully, most of these have default implementations; at minimum, we only need to implement the five members startIndex, endIndex, subscript, index(after:), and index(before:). In this book we'll go a little further than that and also implement forEach and count. When it makes a difference, we'll also add custom implementations for formIndex(after:), and formIndex(before:). For the most part, we'll leave default implementations for everything else, even though we could sometimes write specializations that would work much faster.

# Semantic Requirements

Implementing a Swift protocol usually means more than just conforming to its explicit requirements — most protocols come with an additional set of semantic requirements that aren't expressible in the type system, and these requirements need to be

documented separately. Our `SortedSet` protocol is no different; what follows are five properties we expect all implementations to satisfy.

1. **Ordering:** Elements inside the collection must be kept sorted. To be more specific: if `i` and `j` are both valid and subscriptable indices in some `set` implementing `SortedSet`, then `i < j` must be equivalent to `set[i] < set[j]`. (This also implies that our sets won't have duplicate elements.)
2. **Value semantics:** Mutating an instance of a `SortedSet` type via one variable must not affect the value of any other variable of the same type. Conforming types must behave *as if* each variable held its own unique value, entirely independent of all other variables.
3. **Copy-on-write:** Copying a `SortedSet` value into a new variable must be an $O(1)$ operation. Storage may be partially or fully shared between different `SortedSet` values. Mutations must check for shared storage and create copies when necessary to satisfy value semantics. Therefore, mutations may take longer to complete when storage is shared.
4. **Index specificity:** Indices are associated with a particular `SortedSet` instance; they're only guaranteed to be valid for that specific instance and its unmutated direct copies. Even if `a` and `b` are `SortedSet`s of the same type containing the exact same elements, indices of `a` may not be valid in `b`. (This unfortunate relaxation of true value semantics seems to be technically unavoidable in general.)
5. **Index invalidation:** Any mutation of a `SortedSet` *may* invalidate all existing indices of it, including its `startIndex` and `endIndex`. Implementations aren't *required* to always invalidate every index, but they're allowed to do so. (This point isn't really a requirement, because it's impossible to violate it. It's merely a reminder that unless we know better, we need to assume collection indices are fragile and should be handled with care.)

Note that the compiler won't complain if we forget to implement any of these requirements. But it's still crucial that we implement them so that generic code working with sorted sets has consistent behavior.

If we were writing a real-life, production-ready implementation of sorted sets, we wouldn't need the `SortedSet` protocol at all: we'd simply define a single type that implements all requirements directly. However, we'll be writing several variants of

sorted sets, so it's nice to have a protocol that spells out our requirements and on which we can define extensions that are common to all such types.

Before we even have a concrete implementation of SortedSet, let's get right into defining one such extension!

# Printing Sorted Sets

It's useful to provide a default implementation for description so that we need not spend time on it later. Because all sorted sets are collections, we can use standard collection methods to print sorted sets the same way as the standard library's array or set values — as a comma-separated list of elements enclosed in brackets:

```
extension SortedSet {
  public var description: String {
    let contents = self.lazy.map { "\($0)" }.joined(separator: ", ")
    return "[\(contents)]"
  }
}
```

It's also worth creating a default implementation of customPlaygroundQuickLook so that our collections print a little better in playgrounds. The default Quick Look view can be difficult to understand, especially at a glance, so let's replace it with a simple variant that just sets the description in a monospaced font using an attributed string:

```
#if os(iOS)
import UIKit

extension PlaygroundQuickLook {
  public static func monospacedText(_ string: String) -> PlaygroundQuickLook {
    let text = NSMutableAttributedString(string: string)
    let range = NSRange(location: 0, length: text.length)
    let style = NSParagraphStyle.default.mutableCopy() as! NSMutableParagraphStyle
    style.lineSpacing = 0
    style.alignment = .left
    style.maximumLineHeight = 17
```

```
    text.addAttribute(.font, value: UIFont(name: "Menlo", size: 13)!, range: range)
    text.addAttribute(.paragraphStyle, value: style, range: range)
    return PlaygroundQuickLook.attributedString(text)
  }
}
#endif

extension SortedSet {
  public var customPlaygroundQuickLook: PlaygroundQuickLook {
    #if os(iOS)
      return .monospacedText(String(describing: self))
    #else
      return .text(String(describing: self))
    #endif
  }
}
```

# Sorted Arrays

2

Possibly the most straightforward way to implement SortedSet is by storing the set's elements in an array. This leads to the definition of a simple structure like the one below:

```
public struct SortedArray<Element: Comparable>: SortedSet {
  fileprivate var storage: [Element] = []

  public init() {}
}
```

To help satisfy the protocol requirements, we'll always keep the storage array sorted, hence the name SortedArray.

# Binary Search

To implement insert and contains, we'll need a method that, given an element, returns the position in the storage array that should hold the element.

To do this fast, we need to implement the *binary search algorithm*. This algorithm works by cutting the array into two equal halves, discarding the one that doesn't contain the element we're looking for, and repeating this process until the array is reduced to a single element. Here's one way to do this in Swift:

```
extension SortedArray {
  func index(for element: Element) -> Int {
    var start = 0
    var end = storage.count
    while start < end {
      let middle = start + (end - start) / 2
      if element > storage[middle] {
        start = middle + 1
      }
      else {
        end = middle
      }
    }
```

```
      return start
    }
  }
```

Note that the loop above needs to perform just one more iteration whenever we double the number of elements in the set. That's rather cheap! This is what people mean when they say binary search has *logarithmic complexity*: its running time is roughly proportional to the logarithm of the size of the data. (The big-O notation for such a function is $O(\log n)$.)

Binary search is deceptively short, but it's a rather delicate algorithm that can be tricky to implement correctly. It includes a lot of subtle index arithmetic, and there are ample opportunities for off-by-one errors, overflow problems, or other mistakes. For example, the expression `start + (end - start) / 2` seems like a strange way to calculate the middle index; we'd normally write `(start + end) / 2` instead. However, these two expressions don't always have the same results: the second version contains an addition that may overflow if the collection is huge, thereby resulting in a runtime error.

Hopefully a generic binary search method will get added to the Swift standard library at some point. Meanwhile, if you ever need to implement it, please find a good algorithms book to use as a reference (though I guess this book will do in a pinch too). Don't forget to test your code; sometimes even books have bugs! I find that 100% unit test coverage works great for catching most of my own errors.

Our `index(for:)` function does something similar to `Collection`'s standard `index(of:)` method, except our version always returns a valid index, even if the element isn't currently in the set. This subtle but very important difference will make `index(for:)` usable for insertion too.

# Lookup Methods

Having mentioned `index(of:)`, it's a good idea to define it in terms of `index(for:)` so that it uses the better algorithm too:

```
extension SortedArray {
  public func index(of element: Element) -> Int? {
    let index = self.index(for: element)
```

```
    guard index < count, storage[index] == element else { return nil }
    return index
  }
}
```

The default implementation in Collection executes a linear search by enumerating all elements until it finds the one we're looking for, or until it reaches the end of the collection. This specialized variant can be much, *much* faster than that.

Checking for membership requires a bit less code because we only need to see if the element is there:

```
extension SortedArray {
  public func contains(_ element: Element) -> Bool {
    let index = self.index(for: element)
    return index < count && storage[index] == element
  }
}
```

Implementing forEach is even easier because we can simply forward the call to our storage array. The array is already sorted, so the method will visit elements in the correct order:

```
extension SortedArray {
  public func forEach(_ body: (Element) throws -> Void) rethrows {
    try storage.forEach(body)
  }
}
```

While we're here, it's a good idea to look for other Sequence and Collection members that would benefit from a specialized implementation. For example, sequences with Comparable elements have a sorted() method that returns a sorted array of all elements in the sequence. For SortedArray, this can be implemented by simply returning storage:

```
extension SortedArray {
  public func sorted() -> [Element] {
```

```
    return storage
  }
}
```

# Insertion

To insert a new element into a sorted set, we first find its corresponding index using index(for:) and then check if the element is already there. To maintain the invariant that a SortedSet may not contain duplicates, we only insert the element into the storage if it's not present:

```
extension SortedArray {
  @discardableResult
  public mutating func insert(_ newElement: Element) -> (inserted: Bool,
  ↪  memberAfterInsert: Element)
  {
    let index = self.index(for: newElement)
    if index < count && storage[index] == newElement {
      return (false, storage[index])
    }
    storage.insert(newElement, at: index)
    return (true, newElement)
  }
}
```

# Implementing Collection

Next up, let's implement BidirectionalCollection. Since we're storing everything in a single Array, the easiest way to do this is to simply share indices between SortedArray and its storage. By doing this, we'll be able to forward most collection methods to the storage array, which drastically simplifies our implementation.

Array implements more than BidirectionalCollection: it is in fact a RandomAccessCollection, which has the same API surface but much stricter semantic requirements. RandomAccessCollection requires efficient index arithmetic: we must be

able to both offset an index by any amount and measure the distance between any two indices in constant time.

Since we're going to forward everything to storage anyway, it makes sense for SortedArray to implement the same protocol:

```
extension SortedArray: RandomAccessCollection {
  public typealias Indices = CountableRange<Int>

  public var startIndex: Int { return storage.startIndex }
  public var endIndex: Int { return storage.endIndex }

  public subscript(index: Int) -> Element { return storage[index] }
}
```

This completes the implementation of the SortedSet protocol. Yay!

# Examples

Let's check if it all works:

```
var set = SortedArray<Int>()
for i in (0 ..< 22).shuffled() {
  set.insert(2 * i)
}
▶ set
  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40,
↪  42]

▶ set.contains(42)
  true

▶ set.contains(13)
  false
```

It seems to work just fine. But does our new collection have value semantics?

```
let copy = set
set.insert(13)

▶ set.contains(13)
  true

▶ copy.contains(13)
  false
```

It sure looks like it! We didn't have to do anything to implement value semantics; we got it by the mere fact that SortedArray is a struct consisting of a single array. Value semantics is a composable property: structs with stored properties that all have value semantics automatically behave the same way too.

# Performance

When we talk about the performance of an algorithm, we often use the so-called *big-O notation* to describe how changing the input size affects the running time: $O(1)$, $O(n)$, $O(n^2)$, $O(\log n)$, $O(n \log n)$, and so on. This notation has a precise mathematical definition, but you don't really have to look it up — it's enough to understand that we use this notation as shorthand for classifying the *growth rate* of our algorithms. When we double the size of its input, an $O(n)$ algorithm will take no more than twice the time, but an $O(n^2)$ algorithm might become as much as four times slower. We expect that an $O(1)$ algorithm will run for roughly the same time no matter what input it gets.

We can mathematically analyze our algorithms to formally derive such asymptotic complexity estimates. This analysis provides useful indicators of performance, but it isn't infallible; by its very nature, it relies on simplifications that may or may not match actual behavior for real-world working sets running on actual hardware.

To get an idea of the actual performance of our SortedSet implementations, it's therefore useful to run some benchmarks. For example, here's the code for one possible microbenchmark that measures four basic operations — insert, contains, forEach, and iteration using a for statement — on a SortedArray:

```
func benchmark(count: Int, measure: (String, () -> Void) -> Void) {
  var set = SortedArray<Int>()
```

```
  let input = (0 ..< count).shuffled()
  measure("SortedArray.insert") {
    for value in input {
      set.insert(value)
    }
  }

  let lookups = (0 ..< count).shuffled()
  measure("SortedArray.contains") {
    for element in lookups {
      guard set.contains(element) else { fatalError() }
    }
  }

  measure("SortedArray.forEach") {
    var i = 0
    set.forEach { element in
      guard element == i else { fatalError() }
      i += 1
    }
    guard i == input.count else { fatalError() }
  }

  measure("SortedArray.for-in") {
    var i = 0
    for element in set {
      guard element == i else { fatalError() }
      i += 1
    }
    guard i == input.count else { fatalError() }
  }
}
```

The measure parameter is some function that measures the execution time of the closure that's given to it and files it under the name given as its first parameter. One simple way to drive this benchmark function is to call it in a loop of various sizes, printing measurements as we get them:

```
for size in (0 ..< 20).map({ 1 << $0 }) {
```

```
    benchmark(size: size) { name, body in
        let start = Date()
        body()
        let end = Date()
        print("\(name), \(size), \(end.timeIntervalSince(start))")
    }
}
```

This is a simplified version of the actual Attabench benchmarks I ran to draw the plots below. The real code has a lot more benchmarking boilerplate, but the actual measurements (the code inside the `measure` closures) are exactly the same.

Plotting our benchmark results gets us the chart in figure 2.1. Note that in this chart, we're using logarithmic scales on both axes: moving one notch to the right doubles the number of input values, while moving up by one horizontal line indicates a tenfold increase in execution time.

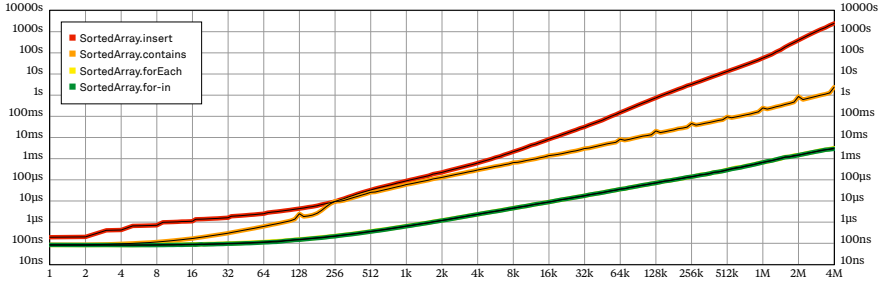



**Figure 2.1: Benchmark results for SortedArray operations, plotting input size vs. overall execution time on a log-log chart.**

Such log-log plots are usually the best way to display benchmarking results. They fit a huge range of data on a single chart without letting large values overwhelm small ones — in this case, we can easily compare execution times from a single element, up to 4 million of them: a difference of 22 binary orders of magnitude!

Additionally, log-log plots make it easy to estimate the actual complexity costs an algorithm exhibits. If a section of a benchmark plot is a straight line segment, then the relationship between input size and execution time can be approximated by a multiple of a simple polynomial function, such as $n$, $n^2$, or even $\sqrt{n}$. The exponent is related to

the slope of the line: the slope of $n^2$ is double that of $n$. With some practice, you'll be able to recognize the most frequently occurring relationships at a glance — there's no need for complicated analysis.

In our case, we know that simply iterating over all the elements inside an array should take $O(n)$ time, and this is confirmed by the plots. `Array.forEach` and the `for-in` loop cost pretty much the same, and after an initial warmup period, they both become straight lines. It takes a little more than three steps to the right for the line to go a single step up, which corresponds to $2^{3.3} \approx 10$, proving a simple linear relationship.

Looking at the plot for `SortedArray.insert`, we find it gradually turns into a straight line at about 4,000 elements, and the slope of the line is roughly double that of `SortedArray.forEach` — so it looks like the execution time of insertion is a quadratic function of the input size. Luckily, this matches our expectations: each time we insert a random element into a sorted array, we have to make space for it by moving (on average) half of the existing elements one slot to the right. So a single insertion is a linear operation, which makes $n$ insertions cost $O(n^2)$.

`SortedArray.contains` does $n$ binary searches, each taking $O(\log n)$ time, so it's supposed to be an $O(n \log n)$ function. It's hard to see this in figure 2.1, but you can verify it if you look close enough: its curve is almost parallel to that of `forEach`, except it subtly drifts upward — it's not a perfectly straight line. One easy way to verify this is by putting the edge of a piece of paper next to the `contains` plot: it curves away from the paper's straight edge, indicating a superlinear relationship.

To highlight the difference between $O(n)$ and $O(n \log n)$ functions, it's useful to divide execution times by the input size, resulting in a chart that displays the average execution time spent on a single element. (I like to call this type of plot an *amortized chart*. I'm not sure the word *amortized* fits this context, but it does sound impressive!) The division eliminates the consistent slope of $O(n)$, making it easier to distinguish linear and logarithmic factors. Figure 2.2 shows such a chart for `SortedArray`. Note how `contains` now has a distinct (if slight) upward slope, while the tail of `forEach` is perfectly flat.

The curve for `contains` has a couple of surprises. First, it has clear spikes at powers-of-two sizes. This is due to an interesting interaction between binary search and the architecture of the level 2 (L2) cache in the MacBook running the benchmarks. The cache is divided into 64-byte *lines*, each of which may hold the contents of main memory from a specific set of physical addresses. By an unfortunate coincidence, if the

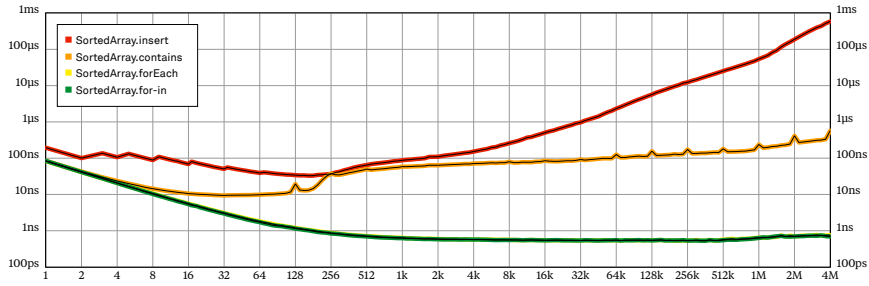



**Figure 2.2: Benchmark results for SortedArray operations, plotting input size vs. average execution time for a single operation on a log-log chart.**

storage size is close to a perfect power of two, successive lookup operations of the binary search algorithm tend to all fall into the same L2 cache line, quickly exhausting its capacity, while other lines remain unused. This effect is called *cache line aliasing*, and it can evidently lead to a quite dramatic slowdown: `contains` takes about twice as long to execute at the top of the spikes than at nearby sizes.

One way to eliminate these spikes is to switch to *ternary search*, dividing the buffer into *three* equal slices on each iteration. An even simpler solution is to perturb binary search, selecting a slightly off-center middle index. To do this, we just need to modify a single line in the implementation of `index(for:)`, adding a small extra offset to the middle index:

```
let middle = start + (end - start) / 2 + (end - start) >> 6
```

This way, the middle index will fall 33/64 the way between the two endpoints, which is enough to prevent cache line aliasing. Unfortunately, the code is now a tiny bit more complicated, and these off-center middle indices generally result in slightly more storage lookups than regular binary search. So the price of eliminating the power-of-two spikes is a small overall slowdown, as demonstrated: by the chart in figure 2.3.

The other surprise of the `contains` curve is that it seems to turn slightly upward at about 64k elements or so. (If you look closely, you may actually detect a similar, although less pronounced, slowdown for `insert`, starting at about a million elements.) At this size, my MacBook's virtual memory subsystem was unable to keep the physical address of all pages of the `storage` array in the CPU's address cache (called the Translation Lookaside Buffer, or TLB for short). The `contains` benchmark randomizes lookups, so its irregular access patterns lead to frequent TLB cache misses, considerably increasing the cost of

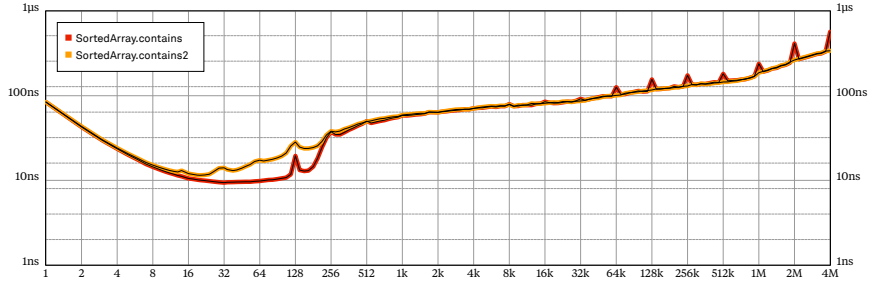



**Figure 2.3: Comparing the performance of binary search (contains) to a variant that prevents cache line aliasing by selecting slightly off-center indices for the middle value (contains2).**

memory accesses. Additionally, as the storage array gets larger, its sheer size overwhelms L1 and L2 caches, and those cache misses contribute a great deal of additional latency.

At the end of the day, it looks like random memory accesses in a large enough contiguous buffer take $O(\log n)$-ish time, not $O(1)$ — so the asymptotic execution time of our binary search is in fact more like $O(\log n \log n)$, and not $O(\log n)$ as we usually believe. Isn't that interesting? (The slowdown disappears if we remove the shuffling of the `lookups` array in the benchmark code above. Try it!)

On the other hand, for small sizes, the `contains` curve keeps remarkably close to `insert`. Some of this is just a side effect of the logarithmic scale: at their closest point, `contains` is still 80% faster than insertion. But the `insert` curve looks surprisingly flat up to about 1,000 elements; it seems that as long as the sorted array is small enough, the average time it takes to insert a new element into it is basically independent of the size of the array. (I believe this is mostly a consequence of the fact that at these sizes, the entire array fits into the L1 cache of the CPU.)

So `SortedArray.insert` seems to be astonishingly fast as long as the array is small. For now, we can file this factoid away as mildly interesting trivia. Keep it in mind though, because this fact will have serious consequences in later parts of this book.

# The Swiftification of NSOrderedSet

3

The Foundation framework includes a class called NSOrderedSet. It's a relatively recent addition, first appearing in 2011 with iOS 5 and OS X 10.7 Lion. NSOrderedSet was added to Foundation specifically in support of ordered relationships in Core Data. It works like a combination of NSArray and NSSet, implementing the API of both classes. It also provides both NSSet's super fast $O(1)$ membership checks and NSArray's speedy $O(1)$ random access indexing. The tradeoff is that it also inherits NSArray's $O(n)$ insertions. Because NSOrderedSet is (most likely) implemented by wrapping an NSSet and an NSArray, it also has a higher memory overhead than either of these components.

NSOrderedSet hasn't been bridged to Swift yet, so it seems like a nice subject for demonstrating how we can define thin wrappers around existing Objective-C classes to bring them closer to the Swift world.

Despite its promising name, NSOrderedSet isn't really a good match for our use case. Although NSOrderedSet does keep its elements ordered, it doesn't enforce any particular ordering relation — you can insert elements in whichever order you like and NSOrderedSet will remember it for you, just like an array. Not having a predefined ordering is the difference between "ordered" and "sorted," and this is why NSOrderedSet is not called NSSortedSet. Its primary purpose is to make lookup operations fast, but it achieves this using hashing rather than comparisons. (There's no equivalent to the Comparable protocol in Foundation; NSObject only provides the functionality of Equatable and Hashable.)

But if NSOrderedSet supports whichever ordering we may choose to use, then it also supports keeping elements sorted according to their Comparable implementation. Clearly, this won't be the ideal use case for NSOrderedSet, but we should be able to make it work. So let's import Foundation and start working on hammering NSOrderedSet into a SortedSet:

```
import Foundation
```

Right off the bat, we run into a couple of major problems.

First, NSOrderedSet is a class, so its instances are reference types. We want our sorted sets to have value semantics.

Second, NSOrderedSet is a heterogeneous sequence — it takes Any members. We could still implement SortedSet by setting its Element type to Any, rather than leaving it as a generic type parameter, but that wouldn't feel like a real solution. What we really want is

a generic homogeneous collection type, with the element type provided by a type parameter.

So we can't just extend NSOrderedSet to implement our protocol. Instead, we're going to define a generic wrapper struct that internally uses an NSOrderedSet instance as storage. This approach is similar to what the Swift standard library does in order to bridge NSArray, NSSet, and NSDictionary instances to Swift's Array, Set, and Dictionary values. So we seem to be heading down the right track.

What should we call our struct? It's tempting to call it NSSortedSet, and it would technically be possible to do so — Swift-only constructs don't depend on prefixes to work around (present or future) naming conflicts. On the other hand, for developers, NS still implies *Apple-provided*, so it'd be impolite and confusing to use it. Let's just go the other way and name our struct OrderedSet. (This name isn't quite right either, but at least it does resemble the name of the underlying data structure.)

```
public struct OrderedSet<Element: Comparable>: SortedSet {
  fileprivate var storage = NSMutableOrderedSet()
}
```

We want to be able to mutate our storage, so we need to declare it as an instance of NSMutableOrderedSet, which is NSOrderedSet's mutable subclass.

# Looking Up Elements

We now have the empty shell of our data structure. Let's begin to fill it with content, starting with the two lookup methods, forEach and contains.

NSOrderedSet implements Sequence, so it already has a forEach method. Assuming elements are kept in the correct order, we can simply forward forEach calls to storage. However, we need to manually downcast values provided by NSOrderedSet to the correct type:

```
extension OrderedSet {
  public func forEach(_ body: (Element) -> Void) {
    storage.forEach { body($0 as! Element) }
```

```
  }
}
```

OrderedSet is in full control of its own storage, so it can guarantee the storage will never contain anything other than an Element. This ensures the forced downcast will always succeed. But it sure is ugly!

NSOrderedSet also happens to provide an implementation for contains, and it seems perfect for our use case. In fact, it's simpler to use than forEach because there's no need for explicit casting:

```
extension OrderedSet {
  public func contains(_ element: Element) -> Bool {
    return storage.contains(element)  // BUG!
  }
}
```

The code above compiles with no warnings, and it appears to work great when Element is Int or String. However, as we already mentioned, NSOrderedSet uses NSObject's hashing API to speed up element lookups. But we don't require Element to be Hashable! How could this work at all?

When we supply a Swift value type to a method taking an Objective-C object — as with storage.contains above, the compiler needs to box the value in a private NSObject subclass that it generates for this purpose. Remember that NSObject has built-in hashing APIs; you can't have an NSObject instance that doesn't support hash. So these autogenerated bridging classes must always have an implementation for hash that's consistent with isEqual(:).

If Element happens to be Hashable, then Swift is able to use the type's own == and hashValue implementations in the bridging class, so Element values get hashed the same way in Objective-C as in Swift, and everything works out perfectly.

However, if Element doesn't implement hashValue, then the bridging class has no choice but to use the default NSObject implementations of both hash and isEqual(_:). Because there's no other information available, these are based on the identity (i.e. the physical address) of the instances, which is effectively random for boxed value types. So two

different bridged instances holding the exact same value will never be considered isEqual(_:) (or return the same hash).

The upshot of all this is that the above code for contains compiles just fine, but it has a fatal bug: if Element isn't Hashable, then it always returns false. Oops!

Oh dear. Lesson of the day: be very, very careful when using Objective-C APIs from Swift. Automatic bridging of Swift values to NSObject instances is extremely convenient, but it has subtle pitfalls. There's nothing in the code that explicitly warns about this problem: no exclamation marks, no explicit cast, nothing.

Well, now we know we can't rely on NSOrderedSet's own lookup methods in our case. Instead we have to look for some alternative APIs to find elements. Thankfully, NSOrderedSet also includes a method that was specifically designed for looking up elements when we know they're sorted according to some comparator function:

```
class NSOrderedSet: NSObject { // in Foundation
  ...
  func index(of object: Any, inSortedRange range: NSRange, options:
  ↪  NSBinarySearchingOptions = [], usingComparator: (Any, Any) -> ComparisonResult)
  ↪  -> Int
  ...
}
```

Presumably this implements some form of binary search, so it should be fast enough. Our elements are sorted according to their Comparable implementation, so we can use Swift's < and > operators to define a suitable comparator function:

```
extension OrderedSet {
  fileprivate static func compare(_ a: Any, _ b: Any) -> ComparisonResult
  {
    let a = a as! Element, b = b as! Element
    if a < b { return .orderedAscending }
    if a > b { return .orderedDescending }
    return .orderedSame
  }
}
```

We can use this comparator to define a method for getting the index for a particular element. This happens to be what Collection's index(of:) method is supposed to do, so let's make sure our definition refines the default implementation of it:

```
extension OrderedSet {
  public func index(of element: Element) -> Int? {
    let index = storage.index(
      of: element,
      inSortedRange: NSRange(0 ..< storage.count),
      usingComparator: OrderedSet.compare)
    return index == NSNotFound ? nil : index
  }
}
```

Once we have this function, contains reduces to a tiny transformation of its result:

```
extension OrderedSet {
  public func contains(_ element: Element) -> Bool {
    return index(of: element) != nil
  }
}
```

I don't know about you, but I found this a lot more complicated than I originally expected. The fine details of how values are bridged into Objective-C *sometimes* have far-reaching consequences that may break our code in subtle but fatal ways. If we aren't aware of these wrinkles, we may be unpleasantly surprised.

NSOrderedSet's flagship feature is that its contains implementation is super fast — so it's rather sad that we can't use it. But there's still hope! Consider that while NSOrderedSet.contains may mistakenly report false for certain types, it never returns true if the value isn't in fact in the set. Therefore, we can write a variant of OrderedSet.contains that still calls it as a shortcut, possibly eliminating the need for binary search:

```
extension OrderedSet {
  public func contains2(_ element: Element) -> Bool {
    return storage.contains(element) || index(of: element) != nil
```

```
    }
}
```

For Hashable elements, this variant will return true faster than index(of:). However, it's slightly slower for values that aren't members of the set and for types that aren't hashable.

# Implementing Collection

NSOrderedSet only conforms to Sequence, and not to Collection. (This isn't some unique quirk; its better-known friends NSArray and NSSet do the same.) Nevertheless, NSOrderedSet does provide some integer-based indexing methods we can use to implement RandomAccessCollection in our OrderedSet type:

```
extension OrderedSet: RandomAccessCollection {
    public typealias Index = Int
    public typealias Indices = CountableRange<Int>

    public var startIndex: Int { return 0 }
    public var endIndex: Int { return storage.count }
    public subscript(i: Int) -> Element { return storage[i] as! Element }
}
```

This turned out to be refreshingly simple.

# Guaranteeing Value Semantics

SortedSet requires value semantics, meaning we want every variable that contains a sorted set to behave as if it holds a unique copy of its value, completely independent of any other variable.

We won't get this for free this time! Our OrderedSet struct consists of a reference to a class instance, so copying an OrderedSet value into another variable will just increment the reference count of the storage object.

This means that two OrderedSet variables may easily share the same storage:

```
var a = OrderedSet()
var b = a
```

The result of the code above is depicted in figure 3.1.

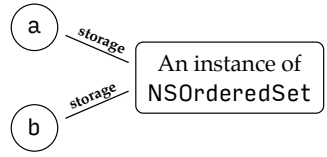



**Figure 3.1: Two OrderedSet values sharing a reference to the same storage object.**

While storage may be shared, mutating methods like insert must nevertheless only modify the set instance that's held by the variable on which they were called. One way to implement this would be to always make a brand new copy of storage before we do anything that modifies it. However, that would be wasteful — it's often the case that the OrderedSet value we have holds the only reference to its storage, and in that case, it'd be safe to modify it directly, without cloning anything.

The Swift standard library provides a function named isKnownUniquelyReferenced that can be called to determine if a particular reference to an object is the sole reference to it. If this function returns true, then we know nobody else holds a reference to the same object, so it's safe to modify it directly.

(Note that this function only looks at strong references; it doesn't count weak and unowned references. So we can't *really* detect every case when somebody might be holding a reference. Luckily, this isn't a problem in our case, because storage is a private property — only code inside OrderedSet has access to it, and we'll never create such "hidden" references. Not counting weak/unowned references is a deliberate feature and not an accidental oversight; it allows indices of more complicated collections to include storage references without forcing COW copies when, say, an element is removed from a certain index. We'll see examples of such index definitions later in this book.)

However, there's still a major gotcha: `isKnownUniquelyReferenced` never returns `true` for subclasses of `NSObject`, because such classes have their own reference count implementation that can't always guarantee a correct result. Of course, `NSOrderedSet` is a subclass of `NSObject`, so it looks like we're screwed. Woe is us!

Oh but wait! If we extend `OrderedSet` to include a reference to some dummy Swift class, we can use this dummy reference to determine the uniqueness of the storage reference too. Copying an `OrderedSet` value will add new references to both its members, so the reference counts of both objects will always remain in sync. So let's modify the definition of `OrderedSet` to include an extra member:

```
private class Canary {}

public struct OrderedSet<Element: Comparable>: SortedSet {
  fileprivate var storage = NSMutableOrderedSet()
  fileprivate var canary = Canary()
  public init() {}
}
```

The sole purpose of `canary` is to indicate whether it's safe to mutate `storage`. (Another way of doing this would be to put the `NSMutableOrderedSet` reference *inside* the new Swift class. That would work just fine too.)

Now we can define a method that ensures our storage is safe to modify:

```
extension OrderedSet {
  fileprivate mutating func makeUnique() {
    if !isKnownUniquelyReferenced(&canary) {
      storage = storage.mutableCopy() as! NSMutableOrderedSet
      canary = Canary()
    }
  }
}
```

Note that we need to create a new canary whenever we find the old one has expired. If we were to forget to do that, this function would continue copying the storage every time it's called.

Now implementing value semantics becomes as simple as remembering to call makeUnique before mutating anything.

# Insertion

Finally, let's do insert. The insert method in NSMutableOrderedSet works like the one in NSMutableArray — it takes an integer index for the new element:

```
class NSOrderedSet: NSObject { // in Foundation
  ...
  func insert(_ object: Any, at idx: Int)
  ...
}
```

Thankfully, the index(of:inSortedRange:options:usingComparator:) method we used above can also be convinced to find the index where a new element should be inserted to maintain the sort order; we just have to set its options parameter to .insertionIndex. This way, it returns a valid index even when the element isn't already in the set:

```
extension OrderedSet {
  fileprivate func index(for value: Element) -> Int {
    return storage.index(
      of: value,
      inSortedRange: NSRange(0 ..< storage.count),
      options: .insertionIndex,
      usingComparator: OrderedSet.compare)
  }
}
```

OK, now we're ready to do actual insertions. We just need to call index(for:) with the new element and check whether the element is already there or not:

```
extension OrderedSet {
  @discardableResult
```

```
  public mutating func insert(_ newElement: Element) -> (inserted: Bool,
  ↪   memberAfterInsert: Element)
  {
    let index = self.index(for: newElement)
    if index < storage.count, storage[index] as! Element == newElement {
      return (false, storage[index] as! Element)
    }
    makeUnique()
    storage.insert(newElement, at: index)
    return (true, newElement)
  }
}
```

We expended a considerable amount of effort implementing `makeUnique`; it'd be a shame if we forgot to call it above. But it's easy to make that mistake and then wonder why inserting a value into one set sometimes modifies some other sets too.

That's all! We now have a second `SortedSet` implementation to play with.

# Let's See If This Thing Works

Here's some code that inserts the numbers between 1 and 20 into a sorted set in random order:

```
var set = OrderedSet<Int>()
for i in (1 ... 20).shuffled() {
  set.insert(i)
}
```

The set is supposed to get the numbers sorted; let's see if it works correctly:

```
▶ set
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Terrific! How about `contains`? Does it find things correctly?

```
▶ set.contains(7)
  true
▶ set.contains(42)
  false
```

We should also be able to use Collection's methods to manipulate sets. For example, let's try calculating the sum of all elements:

```
▶ set.reduce(0, +)
  210
```

OK, but did we get value semantics right?

```
let copy = set
set.insert(42)
▶ copy
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
▶ set
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 42]
```

That seems to work too. Isn't it marvelous?

We had to work extra to make sure our OrderedSet supports Elements that don't implement Hashable, so it makes sense to check that it works correctly now. Here's a simple comparable struct with a single integer property:

```
struct Value: Comparable {
  let value: Int
  init(_ value: Int) { self.value = value }

  static func ==(left: Value, right: Value) -> Bool {
    return left.value == right.value
  }

  static func <(left: Value, right: Value) -> Bool {
    return left.value < right.value
```

```
    }
}
```

When we convert Values into AnyObjects, they get an isEqual implementation that doesn't use ==, and the hash property returns some random-looking value, as promised:

```
let value = Value(42)
let a = value as AnyObject
let b = value as AnyObject
▶ a.isEqual(b)
  false
▶ a.hash
  140717923043328
▶ b.hash
  140717923038080
```

We can plug this type into our previous examples to verify OrderedSet doesn't rely on hashing to work correctly:

```
var values = OrderedSet<Value>()
(1 ... 20).shuffled().map(Value.init).forEach { values.insert($0) }
▶ values.contains(Value(7))
  true
▶ values.contains(Value(42))
  false
```

Yep, it seems to work fine.

It's a good idea to try running your own tests in the playground version of the book. Also try switching to the buggy version of contains to see how it affects the results.

# Performance

Figure 3.2 charts the performance of OrderedSet operations. The most remarkable aspect of this chart is the huge gap between contains and contains2. Evidently

Foundation wasn't kidding about NSOrderedSet.contains being fast: it's about 15–25 times faster than binary search. Too bad it only works for hashable elements…

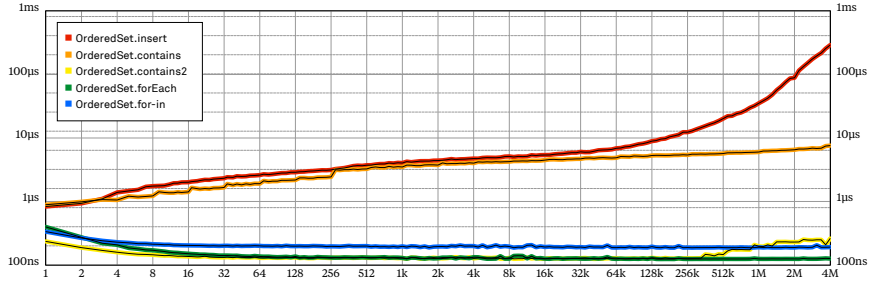



**Figure 3.2: Benchmark results for OrderedSet operations. The plot shows input size vs. average execution time for a single iteration on a log-log chart.**

Interestingly enough, contains2, forEach, and for-in all seem to slow down after 16,000 elements. contains2 looks up random values in a hash table, so it's likely we can attribute its slowdown to cache/TLB thrashing like we did with SortedArray.contains. But this doesn't explain forEach and for-in: they just iterate over elements in the same order they appear in the set, so their curves should be perfectly flat. It's hard to say what's going on without reverse-engineering NSOrderedSet; it's a mystery!

The curve for OrderedSet.insert ends with a quadratic tail, just like SortedArray.insert. Figure 3.3 pits these two insertion implementations against each other. Clearly, NSOrderedSet has a huge overhead compared to Array — at small element counts, the latter can be as much as 64 times faster. (Some of this is due to NSOrderedSet's need for boxing up elements into an NSObject-derived element type; switching the element type from Int to a simple wrapper class around an integer value reduces the gap between the two algorithms to just 800%.) But after about 300,000 elements, NSOrderedSet finally overcomes this disadvantage, and it actually finishes about two times faster than Swift arrays!

What's going on here? The standard library's definition for Array has extensive semantic annotations to help the compiler optimize code in ways not otherwise possible; the compiler is also able to inline Array methods to eliminate even the tiny cost of a function call and to detect further opportunities for optimization. How can some puny, unoptimizable Objective-C class be faster than Array is at inserting elements?

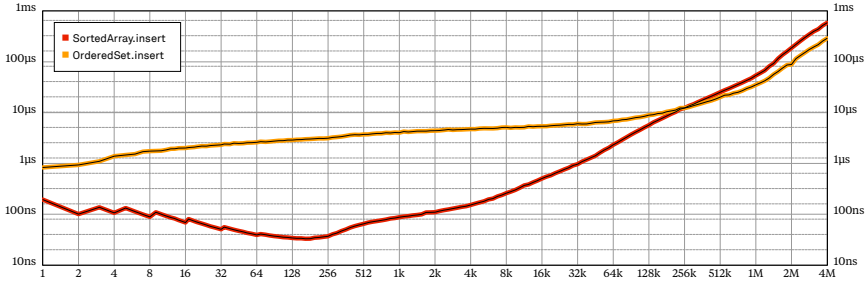



**Figure 3.3: Comparing the performance of two insert implementations.**

Well, the secret is that NSMutableOrderedSet is probably built on top of NSMutableArray, which is not an array at all. It's a double-ended queue, or *deque* for short. Prepending a new element to the front of an NSMutableArray takes the same, constant amount of time as appending one to the end of it. This is in stark contrast to Array.insert, where inserting an element at position 0 is an $O(n)$ operation; it needs to make room for the new element by moving every element one slot to the right.

By sliding elements toward either the front or the back, NSMutableArray.insert never needs to move more than half of the elements; thus, when we have enough elements for insertion time to be dominated by the task of moving stuff, NSMutableArray.insert is, on average, twice as fast as Array.insert, despite the Swift compiler's clever optimizations. That's cool!

Overall though, this 200% speedup doesn't make up for NSOrderedSet's slowness at smaller sizes. Plus, insertion still has the same $O(n^2)$ growth rate, which isn't impressive at all: creating an OrderedSet with 4 million elements takes more than 26 minutes. This may be better than SortedArray (which takes a little less than 50 minutes to do the same), but it's nothing to write home about.

We've put a lot of effort into NSOrderedSet, but the resulting code seems overly complicated, fragile, and slow. Still, this chapter isn't a *complete* failure — we did create another correct implementation of SortedSet, and we've learned a great deal about writing Swift wrappers around legacy Objective-C interfaces, which is a useful skill to have.

Is there a way to implement SortedSet that's meaningfully faster than SortedArray? Well, of course! But to do that, we need to learn about search trees.

# Red-Black Trees

4

*Self-balancing binary search trees* provide efficient algorithms for implementing sorted collections. In particular, these data structures can be used to implement sorted sets so that the insertion of any element only takes logarithmic time. This is a really attractive feature — remember, our `SortedArray` implemented insertions in linear time.

The expression, "self-balancing binary search tree," looks kind of technical. Each word has a specific meaning, so I'll quickly explain them.

A *tree* is a data structure with data stored inside *nodes*, arranged in a branching, tree-like structure. Each tree has a single node at the top called the *root node*. (The root is put on the top because computer scientists have been drawing trees upside down for decades. This is because it's more practical to draw tree diagrams this way, and not because they don't know what an actual tree looks like. Or at least I hope so, anyway.) If a node has no children, then it's called a *leaf node*; otherwise, it's an *internal node*. A tree usually has many leaf nodes.

In general, internal nodes may have any number of child nodes, but in *binary trees*, nodes may only have two children, called *left* and *right*. Some nodes may have two children, while some may have only a left child, only a right child, or no child at all.

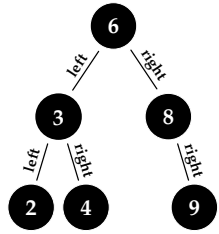



**Figure 4.1: A binary search tree. Node 6 is the root node and nodes 6, 3, and 8 are internal nodes, while nodes 2, 4, and 9 are leaves.**

In *search trees*, the values inside nodes are comparable in some way, and nodes are arranged such that for every node in the tree, all values inside the left subtree are less than the value of the node, and all values to the right are greater than it. This makes it easy to find any particular element.

By *self-balancing*, we mean that the data structure has some sort of mechanism that guarantees the tree remains nice and bushy with a short height, no matter which values it contains and in what order the values are inserted. If the tree was allowed to grow lopsided, even simple operations could become inefficient. (As an extreme example, a tree where each node has at most one child works like a linked list — not very efficient at all.)

There are many ways to create self-balancing binary trees; in this section, we're going to implement the variant called *red-black trees*. This particular flavor needs to store a single extra bit of information in every node to implement the self-balancing part. That extra bit is the node's color, which can be either red or black.

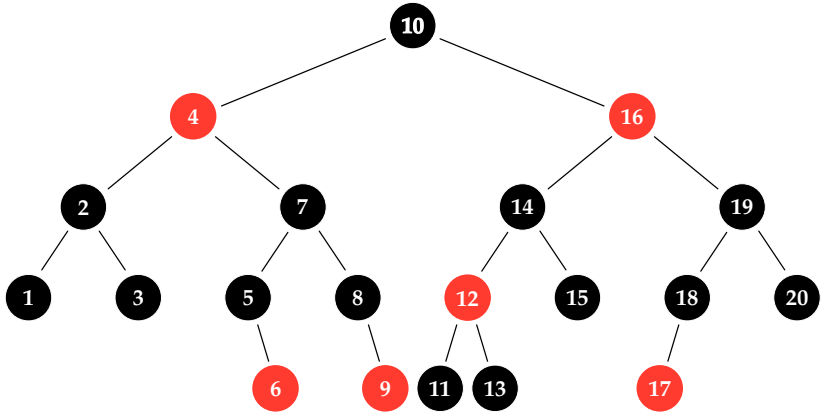



**Figure 4.2: An example red-black tree.**

A red-black tree always keeps its nodes arranged and colored so that the following properties are all true:

1. The root node is black.
2. Red nodes only have black children. (If they have any, that is.)
3. Every path from the root node to an empty spot in the tree contains the same number of black nodes.

Empty spots are places where new nodes can be added to the tree, i.e. where a node has no left or right child. To grow a tree one node larger, we just need to replace one of its empty spots with a new node.

The first property is there to make our algorithms a little bit simpler; it doesn't really affect the tree's shape at all. The last two properties ensure that the tree remains nicely dense, where no empty spot in the tree is more than twice as far from the root node as any other.

To fully understand these balancing properties, it's useful to play with them a little, exploring their edge cases. For example, it's possible to construct a red-black tree that consists solely of black nodes; the tree in figure 4.3 is one example.

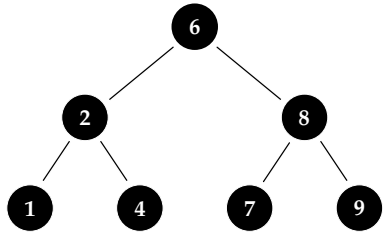



**Figure 4.3: An example red-black tree where every node is black.**

If we try to construct more examples, we'll soon realize that the third red-black property restricts such trees into a particular shape where all internal nodes have two children and all leaf nodes are on the same level. Trees with this shape are called *perfect trees* because they're so perfectly balanced and symmetrical. This is the ideal shape all balanced trees aspire to, because it puts every node as close to the root as possible.

Unfortunately, it'd be unreasonable to require balancing algorithms to maintain perfect trees: indeed, it's only possible to construct perfect binary trees with certain node counts. There's no perfect binary tree with four nodes, for example.

To make red-black trees practical, the third red-black property therefore uses a weaker definition of balancing, where red nodes aren't counted. However, to make sure things don't get *too* unruly, the second property limits the number of red nodes to something

reasonable: it ensures that on any particular path in the tree, the number of red nodes will never exceed the number of black ones.

# Algebraic Data Types

Now that we have a rough idea of what red-black trees are supposed to be, let's get right into their implementation, starting with a node's color. This information is traditionally tucked inside an unused bit in the node's binary representation using low-level hacks so that it doesn't take up extra space. But we like clean, safe code, so we'll represent the color using two cases of a regular enumeration:

```
public enum Color {
  case black
  case red
}
```

> The Swift compiler may sometimes be able to tuck the color value into such an unused bit on its own, without us having to do anything. The binary layout of Swift types is much more flexible than in C/C++/Objective-C, and the compiler has considerable freedom in deciding how to pack things up. This is especially the case for enums with associated values, where the compiler is often able to find unused bit patterns to represent cases without allocating extra space for distinguishing between them. For example, an `Optional` wrapping a reference type fits into the same space as the reference itself. The `Optional.none` case is represented by an (all-zero) bit pattern that's never used by any valid reference. (Incidentally, the same all-zero bit pattern represents the null pointer in C and the `nil` value in Objective-C, allowing some measure of binary compatibility.)

A tree itself is either empty, or it has a root node with a color, a value, and two children: left and right. Swift allows enum cases to include field values, which makes it possible to translate this description into a second enum type:

```
public enum RedBlackTree<Element: Comparable> {
  case empty
```

```
    indirect case node(Color, Element, RedBlackTree, RedBlackTree)
}
```

In real code, we'd usually give labels to the fields of a node so that their roles are clear. We keep them unnamed here only to prevent unfortunate line breaks in the code samples. Sorry about that.

We have to use the `indirect case` syntax because the children of a node are trees themselves. The `indirect` keyword highlights the presence of recursive nesting to readers of our code, and it also allows the compiler to box node values into some hidden heap-allocated reference type. (It's necessary to do this to prevent trouble, like the compiler not being able to assign a specific storage size to values of the enum. Recursive data structures are tricky.)

Enums with fields like this are Swift's way of defining *algebraic data types*, which (as we'll soon see) provide a particularly powerful and elegant way of building data structures.

Having defined the skeleton of our new red-black tree type, we're now ready to move on to implementing `SortedSet` methods on it.

# Pattern Matching and Recursion

When working with algebraic data types, we typically use pattern matching to break down the problem we want to solve into a number of distinct cases. Then we solve each of these cases one by one, often relying on recursion to solve a particular case in terms of the solution of a slightly smaller variant of the same problem.

## contains

For example, let's see an implementation for `contains`. Our search tree stores values in a specific order, and we can rely on this to split the problem of finding an element into four smaller cases:

1. If the tree is empty, then it doesn't contain any elements, so `contains` must return `false`.

2. Otherwise, the tree must have a root node at the top. If the value stored there happens to equal the element we're looking for, then we know that the tree contains the element; `contains` should return `true` in this case.
3. Otherwise, if the root value is greater than the element we're looking for, then the element is definitely not inside the right subtree. The tree contains the element if and only if the left subtree contains it.
4. Otherwise, the root value is less than the element, so we only need to look inside the right subtree.

We can translate this recipe directly into Swift using a `switch` statement:

```
public extension RedBlackTree {
  func contains(_ element: Element) -> Bool {
    switch self {
    case .empty:
      return false
    case .node(_, element, _, _):
      return true
    case let .node(_, value, left, _) where value > element:
      return left.contains(element)
    case let .node(_, _, _, right):
      return right.contains(element)
    }
  }
}
```

Swift's pattern matching syntax is a natural way to express these kinds of structural conditions. Most methods we define on `RedBlackTree` will have this exact same structure, although the details will of course be different.

## forEach

In `forEach`, we want to call a closure on all elements inside the tree in ascending order. This is easy to do if the tree is empty: we don't need to do anything. Otherwise, we need to visit all elements inside the left subtree, then the element stored in the root node, and finally all elements inside the right subtree.

This all lends itself nicely to another recursive method that uses a switch statement:

```
public extension RedBlackTree {
  func forEach(_ body: (Element) throws -> Void) rethrows {
    switch self {
    case .empty:
      break
    case let .node(_, value, left, right):
      try left.forEach(body)
      try body(value)
      try right.forEach(body)
    }
  }
}
```

This example turned out to be even shorter than `contains`, but you can see it shares the same structure: a few pattern matching cases in a `switch` statement, flavored with occasional uses of recursion in the case bodies.

This algorithm is doing a so-called *inorder walk* on the tree. It's useful to know this term; it may pop up at any time during polite dinner table conversation, provided you dine with the same kind of friends as I do. When we do an inorder walk in a search tree, we visit elements "in order," from smallest to largest.

# Tree Diagrams

Next, we'll take a little detour to do an interesting custom implementation for `CustomStringConvertible`. (As I hope you remember, we've already written a generic version in the introduction.)

When working on a new data structure, it's usually worth investing a little extra time to make sure we're able to print out the exact structure of our values in an easily understandable format. The investment usually pays back in spades by making debugging a lot easier. For red-black trees, we're going to *really* make an effort and use Unicode art to construct elaborate little tree diagrams.

We start with a property that returns a symbolic representation for Color. Little black and white squares will do the job just fine:

```
extension Color {
  var symbol: String {
    switch self {
    case .black: return "■"
    case .red:   return "□"
    }
  }
}
```

We'll use a clever adaptation of the forEach function to generate the diagram itself. To change things up a bit, I'll leave it up to you to figure out how it works. I promise it's not as tricky as it looks! (Hint: experiment with changing some of the string literals to see what changes in the resulting output below.)

```
extension RedBlackTree: CustomStringConvertible {
  func diagram(_ top: String, _ root: String, _ bottom: String) -> String {
    switch self {
    case .empty:
      return root + "•\n"
    case let .node(color, value, .empty, .empty):
      return root + "\(color.symbol) \(value)\n"
    case let .node(color, value, left, right):
      return right.diagram(top + "    ", top + " ┌──", top + " │  ")
        + root + "\(color.symbol) \(value)\n"
        + left.diagram(bottom + " │  ", bottom + " └──", bottom + "    ")
    }
  }

  public var description: String {
    return self.diagram("", "", "")
  }
}
```

Let's see what diagram does for some simple tree values. RedBlackTree is just an enum, so we can construct arbitrary trees by manually nesting enum cases.

1. An empty tree is printed as a small black dot. This is implemented by the first case pattern in the diagram method above:

```
let emptyTree: RedBlackTree<Int> = .empty
▶ emptyTree
  •
```

2. A tree with a single node matches the second case pattern, so it's printed on a single line consisting of its color and value:

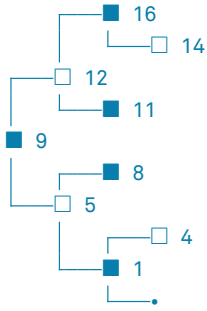
```
let tinyTree: RedBlackTree<Int> = .node(.black, 42, .empty, .empty)
▶ tinyTree
  ■ 42
```

3. Finally, a tree with a root containing non-empty children matches the third case. Its description looks similar to the previous case, except the root node has grown left and right limbs displaying its children:

```
let smallTree: RedBlackTree<Int> =
  .node(.black, 2,
    .node(.red, 1, .empty, .empty),
    .node(.red, 3, .empty, .empty))
▶ smallTree
  ┌──□ 3
  ■ 2
  └──□ 1
```

Even more complicated trees are displayed, well, like trees:

```
let bigTree: RedBlackTree<Int> =
  .node(.black, 9,
    .node(.red, 5,
      .node(.black, 1, .empty, .node(.red, 4, .empty, .empty)),
      .node(.black, 8, .empty, .empty)),
    .node(.red, 12,
      .node(.black, 11, .empty, .empty),
      .node(.black, 16,
        .node(.red, 14, .empty, .empty),
        .node(.red, 17, .empty, .empty))))
▶ bigTree
            ┌──□ 17
```




Isn't this neat? Working with algebraic data types can be so effortless that you sometimes don't even realize you're doing something that's supposed to be complicated.

We now return to our regularly scheduled programming. Let's continue building our sorted set implementation.

# Insertion

In SortedSet, we defined insertion as a mutating function. For the case of our red-black trees, however, it's going to be much easier to define a functional variant that returns a brand-new tree instead of modifying the existing one. Here's a suitable signature for it:

```
func inserting(_ element: Element) -> (tree: RedBlackTree, existingMember: Element?)
```

Given a function like this, we can implement mutating insertion by assigning the tree it returns to self:

```
extension RedBlackTree {
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪ Element)
  {
    let (tree, old) = inserting(element)
    self = tree
    return (old == nil, old ?? element)
```

```
  }
}
```

For inserting, we're going to implement a beautiful pattern matching algorithm first published by Chris Okasaki in 1999.

Remember that red-black trees need to satisfy three requirements:

1. The root node is black.
2. Red nodes only have black children. (If they have any, that is.)
3. Every path from the root node to an empty spot in the tree contains the same number of black nodes.

One way to ensure the first requirement is to just insert the element as if the requirement didn't exist. If the result happens to come out with a red root, then it's easy enough to paint it black, and doing this won't affect any of the other requirements. (The second requirement only cares about red nodes, so we could paint every node black, and it'd still hold. As for the third requirement: the root node is on every path inside the tree, so painting it black will consistently increment the number of black nodes on every path by one. Therefore, if the tree satisfied the third requirement before we painted its root black, then the repainted tree won't violate it either.)

All this consideration leads us to define inserting as a short wrapper function dedicated to ensuring the first requirement. It delegates the actual insertion to some internal helper function we have yet to define:

```
extension RedBlackTree {
  public func inserting(_ element: Element) -> (tree: RedBlackTree, existingMember:
  ↪ Element?) {
    let (tree, old) = _inserting(element)
    switch tree {
    case let .node(.red, value, left, right):
      return (.node(.black, value, left, right), old)
    default:
      return (tree, old)
    }
  }
}
```

Let's tackle this _insertion method next. Its primary job is to find the correct place where the specified element can be inserted into the tree as a new leaf node. This task is similar to contains, because we need to follow the same path down the tree as we would when looking for an existing element. So it shouldn't come as a shock to find that _insertion and contains have the exact same structure; they just need to return *slightly* different things in each case statement.

Let's go through each of the four cases one by one, explaining the code as we go:

```
extension RedBlackTree {
  func _inserting(_ element: Element) -> (tree: RedBlackTree, old: Element?)
  {
    switch self {
```

First, to insert a new element into an empty tree, we simply need to create a root node with the specified value:

```
    case .empty:
      return (.node(.red, element, .empty, .empty), nil)
```

Notably, this violates the first requirement, since we're creating a tree with a red root. But that's not a problem, since inserting will fix this after we return. Additionally, the other two requirements are fine.

Let's move on to the second case! If the root node has the same value as the one we're trying to insert, then the tree already contains it, so we can safely return self, along with a copy of the existing member:

```
    case let .node(_, value, _, _) where value == element:
      return (self, value)
```

Since self was supposed to satisfy all requirements, returning it unmodified won't break them either.

Otherwise, the value should be inserted into either the left or right subtree of the root node, depending on how it compares to the root value — so we need to do a recursive

call. If the return value indicates the value wasn't already in the tree, then we need to return a copy of our root node, where the previous version of the same subtree is replaced by this new one. (Otherwise, we just return self again.)

```
      case let .node(color, value, left, right) where value > element:
        let (l, old) = left._inserting(element)
        if let old = old { return (self, old) }
        return (balanced(color, value, l, right), nil)

      case let .node(color, value, left, right):
        let (r, old) = right._inserting(element)
        if let old = old { return (self, old) }
        return (balanced(color, value, left, r), nil)
      }
    }
}
```

The two cases above make calls to a mysterious balanced function before returning the new tree. This function is where the red-black magic happens. (The _inserting function is otherwise mostly identical to how we'd define insertion into a common everyday binary search tree; there's very little in it that's specific to red-black trees.)

# Balancing

The job of the balanced method is to detect if a balancing requirement is broken, and if so, to fix the tree by cleverly rearranging its nodes and returning a tree that satisfies the criteria.

The _inserting code above creates new nodes as red leaf nodes; in most cases, this is done in a recursive call, the result of which is then inserted as a child of an existing node. Which red-black requirements may be broken by this?

The first requirement is safe, since it's about the root node, not a child (and we take care of the root in inserting, so here we can safely ignore it anyway). Since the third property doesn't care about red nodes, inserting a new red leaf node won't ever break it either. However, nothing in the code prevents us from inserting a red node under a red parent, so we may end up violating the second property.

So balanced only needs to check the second requirement and somehow fix it without breaking the third. In a valid red-black tree, a red node always has a black parent; so when insertion breaks the second requirement, the result always matches one of the cases (1–4) in figure 4.4.

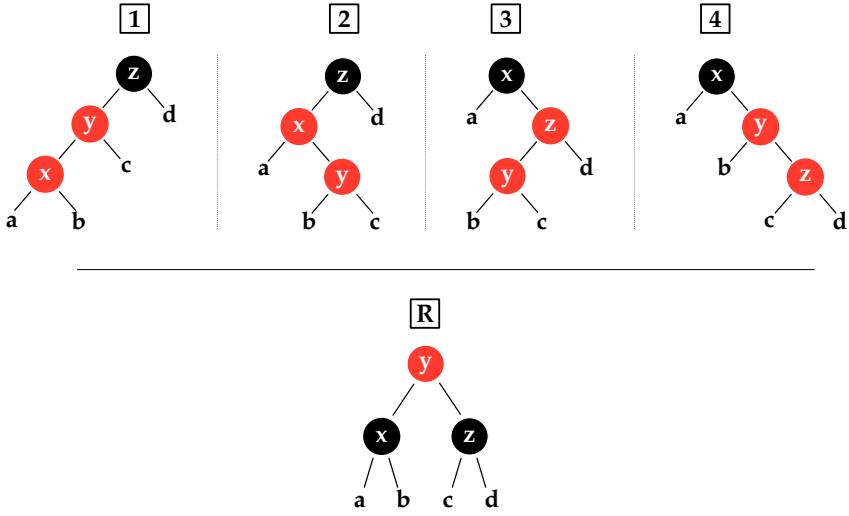



**Figure 4.4: All four situations in which a red node may have a red child after inserting a single element. The labels *x*, *y*, and *z* stand for values, while *a*, *b*, *c*, and *d* are (possibly empty) subtrees. If the tree matches any of the patterns 1--4, then its nodes need to be reorganized into the pattern R.**

We can implement rebalancing by detecting if our tree matches one of these patterns after insertion, and if so, reorganize the tree so that it matches pattern R. This resulting pattern cleverly restores the second requirement without breaking the third.

Can you guess which tool we're going to use to implement this?

By some sort of amazing coincidence, this particular problem is an excellent demonstration of the power of pattern matching on algebraic data types. Let's start by turning the five diagrams in figure 4.4 into Swift expressions. When we created our fancy Unicode tree descriptions, we saw how carefully constructed Swift expressions can be

used to build small trees; now we just need to apply this knowledge to the following diagrams:

```
1: .node(.black, z, .node(.red, y, .node(.red, x, a, b), c), d)
2: .node(.black, z, .node(.red, x, a, .node(.red, y, b, c)), d)
3: .node(.black, x, a, .node(.red, z, .node(.red, y, b, c), d))
4: .node(.black, x, a, .node(.red, y, b, .node(.red, z, c, d)))
R: .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
```

At this point, we're essentially done! To get a correct implementation of the balanced function, we just need to take these expressions and bolt them onto a switch statement:

```
extension RedBlackTree {
  func balanced(_ color: Color, _ value: Element, _ left: RedBlackTree, _ right:
  ↪  RedBlackTree) -> RedBlackTree {
    switch (color, value, left, right) {
    case let (.black, z, .node(.red, y, .node(.red, x, a, b), c), d),
      let (.black, z, .node(.red, x, a, .node(.red, y, b, c)), d),
      let (.black, x, a, .node(.red, z, .node(.red, y, b, c), d)),
      let (.black, x, a, .node(.red, y, b, .node(.red, z, c, d))):
      return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
    default:
      return .node(color, value, left,  right)
    }
  }
}
```

This is a remarkable way of programming. In essence, we translated figure 4.4 directly into Swift in such a way that the five diagrams we used to explain the task are still recognizable in the code!

There's a slight problem though. While the above is perfectly valid Swift code, unfortunately it crashes version 4 of the Swift compiler. To work around this bug, we need to handle each case separately, repeating the exact same body four times:

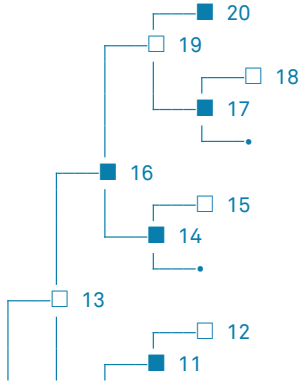
```
extension RedBlackTree {
  func balanced(_ color: Color, _ value: Element, _ left: RedBlackTree, _ right:
  ↪  RedBlackTree) -> RedBlackTree {
```

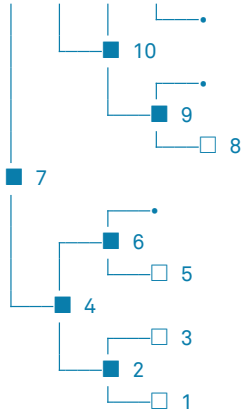
```
      switch (color, value, left, right) {
      case let (.black, z, .node(.red, y, .node(.red, x, a, b), c), d):
        return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
      case let (.black, z, .node(.red, x, a, .node(.red, y, b, c)), d):
        return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
      case let (.black, x, a, .node(.red, z, .node(.red, y, b, c), d)):
        return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
      case let (.black, x, a, .node(.red, y, b, .node(.red, z, c, d))):
        return .node(.red, y, .node(.black, x, a, b), .node(.black, z, c, d))
      default:
        return .node(color, value, left, right)
      }
    }
}
```

This workaround dilutes the concentrated wonder of the original a little bit, but thankfully the magic still shines through. (And hopefully the bug will be fixed in a future compiler release.)

To see if insertion actually works, let's create a red-black tree containing numbers from 1 to 20:

```
var set = RedBlackTree<Int>.empty
for i in (1 ... 20).shuffled() {
  set.insert(i)
}
▶ set
```






It all seems to work fine! Hurray! We can see that the tree we get satisfies all three red-black tree properties.

We shuffle the numbers before inserting them, so we get a brand-new variant of the tree each time we execute this code. The book's playground is perfect for trying this.

# Collection

Implementing protocols like Collection is where algebraic data types start to get a little inconvenient to work with. We have to define a suitable index type, and the easiest way to do this is to just use an element itself as the index, like so:

```
extension RedBlackTree {
  public struct Index {
    fileprivate var value: Element?
  }
}
```

The value is an optional because we'll use the nil value to represent the end index.

Collection indices must be comparable. Thankfully, our elements are comparable too, which makes it easy to implement this requirement. The only tricky thing is that we must ensure the end index compares *greater* than anything else:

```
extension RedBlackTree.Index: Comparable {
  public static func ==(left: RedBlackTree<Element>.Index, right:
  ↪  RedBlackTree<Element>.Index) -> Bool {
    return left.value == right.value
  }

  public static func <(left: RedBlackTree<Element>.Index, right:
  ↪  RedBlackTree<Element>.Index) -> Bool {
    if let lv = left.value, let rv = right.value {
      return lv < rv
    }
    return left.value != nil
  }
}
```

Next, we implement specialized versions of the min() and max() extension methods of Sequence to retrieve the smallest and largest elements of a tree. We'll need these below to implement index advancement.

The smallest element is the value stored in the leftmost node; here we use pattern matching and recursion to find it:

```
extension RedBlackTree {
  func min() -> Element? {
    switch self {
    case .empty:
      return nil
    case let .node(_, value, left, _):
      return left.min() ?? value
    }
  }
}
```

Finding the largest element can be done in a similar way. However, to keep things interesting, here's a version of max() that rolls the recursion into a loop:

```
extension RedBlackTree {
  func max() -> Element? {
```

```
    var node = self
    var maximum: Element? = nil
    while case let .node(_, value, _, right) = node {
      maximum = value
      node = right
    }
    return maximum
  }
}
```

Notice how much more difficult it is to understand this code, compared to `min()`. Instead of using a simple expression to define the result, this version uses a `while` loop that keeps mutating some internal state. To see how it works, you have to run the code in your mind, figuring out how `node` and `maximum` change as the loop progresses. But fundamentally, these are just two different ways of expressing the same algorithm, and it makes sense to get used to both. The iterative version can sometimes run slightly faster, so by using it we can trade a little bit of readability in exchange for a little bit of additional performance. But it's usually easier to start with a recursive solution, and we can always rewrite the code if benchmarks tell us it'd be worth the price of decreased readability.

Now that we have the `min()` and `max()` methods, we can start implementing `Collection`.

Let's begin with the basics: `startIndex`, `endIndex`, and `subscript`:

```
extension RedBlackTree: Collection {
  public var startIndex: Index { return Index(value: self.min()) }
  public var endIndex: Index { return Index(value: nil) }

  public subscript(i: Index) -> Element {
    return i.value!
  }
}
```

`Collection` implementations should always document their rules about index invalidation. In our case, we could formally specify that an index, `i`, that was originally created for tree, `t`, is a *valid index* in some other tree, `u`, if and only if either `i` is the end index or the value `t[i]` is also contained in `u`. This definition allows people to reuse some

indices after certain tree mutations, which can be a useful feature. (This rule is different than, say, the way Array indices work, because in our case, the index is based on the value, not the position, of a particular member. This is a slightly unusual way to define index invalidation, but it doesn't violate Collection's semantic requirements.)

Our subscript implementation is particularly short because it simply unwraps the value inside the index. However, this isn't very nice, because it doesn't verify that the value stored in the index is actually inside the tree. We can subscript any tree with any index, and we'll get a result that may or may not make sense. While Collection implementations are free to set their own rules about index invalidation, they should make some effort to verify that those rules are followed at runtime. Subscripting a collection with an invalid index is a serious coding error best handled by trapping rather than silently returning some weird value. But this will have to do for now, and I promise we'll do better from here on out.

Unfortunately, startIndex calls min(), which is a logarithmic operation. This is a problem because Collection requires startIndex and endIndex implementations to finish in constant time. One relatively simple way to fix this would be to cache the minimum value in or near the root of the tree, by, say, introducing a simple wrapper struct around the tree value. (I'll leave implementing this as an exercise for you, Dear Reader. Or we could just move on and pretend this paragraph never happened!)

Anyway, we now need to implement index(after:), which reduces to finding the smallest element inside the tree that's larger than a particular value. We'll write a utility function to do this. To keep our promise about index validation, the function will also return a boolean value indicating whether or not the element originally stored in the index is inside the tree so that we can set a precondition for it:

```
extension RedBlackTree: BidirectionalCollection {
  public func index(after i: Index) -> Index {
    let v = self.value(following: i.value!)
    precondition(v.found)
    return Index(value: v.next)
  }
}
```

The value(following:) function is yet another subtle variation on the theme of contains. Its logic is quite tricky though, and it deserves a second look:

```
extension RedBlackTree {
  func value(following element: Element) -> (found: Bool, next: Element?) {
    switch self {
      case .empty:
        return (false, nil)
      case .node(_, element, _, let right):
        return (true, right.min())
      case let .node(_, value, left, _) where value > element:
        let v = left.value(following: element)
        return (v.found, v.next ?? value)
      case let .node(_, _, _, right):
        return right.value(following: element)
    }
  }
}
```

The trickiest part is what happens in case 3, when the element is inside the left subtree. Usually the element following it is in the same subtree, so a recursive call will return the correct result. But when element is the maximum value in left, the call returns (true, nil). In this case, we need to return the value following the left subtree, which is the value stored in the parent node itself.

We also need to be able to go the other way to find the preceding value. To keep things interesting, we'll do this variant using iteration rather than recursion:

```
extension RedBlackTree {
  func value(preceding element: Element) -> (found: Bool, next: Element?) {
    var node = self
    var previous: Element? = nil
    while case let .node(_, value, left, right) = node {
      if value > element {
        node = left
      }
      else if value < element {
        previous = value
        node = right
      }
      else {
        return (true, left.max() ?? previous)
```

```
        }
      }
      return (false, previous)
    }
  }
```

Note how the purpose of the variable previous here is equivalent to the application of the nil-coalescing operator ?? in the original tricky case above. (Such post-processing of recursive results can't be directly converted into iterative code, but we can always eliminate post-processing by adding new parameters to the function so that it can do the processing before returning from the recursive call. These extra parameters become variables like previous in the iterative code.)

To complete the implementation of BidirectionalCollection, we just need to add a method that calls value(preceding:) to find the predecessor of an index:

```
extension RedBlackTree {
  public func index(before i: Index) -> Index {
    if let value = i.value {
      let v = self.value(preceding: value)
      precondition(v.found)
      return Index(value: v.next)
    }
    else {
      return Index(value: max()!)
    }
  }
}
```

Defining index(after:) and index(before:) this way is definitely more complicated than in our previous efforts. The end result will also probably be quite slow at runtime — all these recursive lookups make the algorithms cost $O(\log n)$ time instead of the more typical $O(1)$. This doesn't violate any Collection rules; it just makes iteration using indexing slower than usual.

Finally, let's see an implementation for count; it's yet another exciting opportunity for practicing pattern matching and recursion:

```
extension RedBlackTree {
  public var count: Int {
    switch self {
    case .empty:
      return 0
    case let .node(_, _, left, right):
      return left.count + 1 + right.count
    }
  }
}
```

If we were to forget to specialize `count`, its default implementation would count the number of steps between `startIndex` and `endIndex`, and that would be considerably slower than this — $O(n \log n)$ vs. our $O(n)$. But this implementation is nothing to write home about either: it still has to visit every node in the tree.

> We could speed this up by having all nodes remember the number of elements under them, which would turn our tree into a so-called *order statistic tree*. (We'd need to update insertion and any other mutations to carefully maintain this extra information. Using this kind of extension is called *augmenting the tree*. Adding element counts is just one of many useful ways a tree can be augmented.)
>
> Order statistic trees provide a number of interesting benefits aside from speeding up `count`. For example, it's possible to find the $i$th smallest/largest value in such a tree in just $O(\log n)$ steps, for any $i$. However, this doesn't come for free: we'd pay for it with increased memory overhead and code complexity.

And there you have it, `RedBlackTree` is now a `BidirectionalCollection`, complete with all the extensions we know and love from stdlib:

```
▶ set.lazy.filter { $0 % 2 == 0 }.map { "\($0)" }.joined(separator: ", ")
  2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

To finish things off, we'll implement `SortedSet`. Luckily, we've already fulfilled all its requirements except for the empty initializer — and adding that one is no big deal either:

```
extension RedBlackTree: SortedSet {
  public init() {
    self = .empty
  }
}
```

All done! For (almost) finishing this chapter, you deserve a challenge: try implementing SetAlgebra's remove(_:) operation. Pro tip: ignore red-black tree requirements in the first version you write. Once you have working code, start thinking about how to rebalance the tree after a deletion. (It's fair game to look it up on the web or in your favorite algorithm book; it's challenging enough to implement this stuff while looking at a reference!)

# Performance

We expect most operations on RedBlackTree to take $O(\log n)$ time, except forEach, which should be $O(n)$ — it performs one recursive call for each node (and empty slot) inside the tree. Executing our usual benchmarks proves this to be largely correct; figure 4.5 plots the results produced on my Mac.

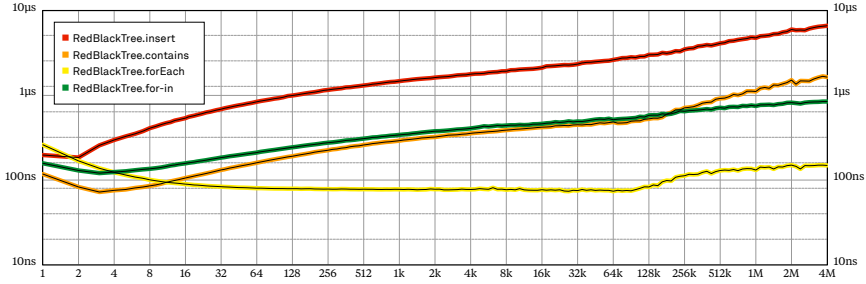



**Figure 4.5: Benchmark results for RedBlackTree operations. The plot shows input size vs. average execution time for a single iteration on a log-log chart.**

The curves are rather noisy. This is because red-black trees allow quite a large variation on the shape of the tree. The depth of the tree depends not just on the number of elements, but also (to a much lesser extent) on the (random) order we insert them. The

performance of most operations is largely proportional to the tree's depth, so we have shaky curves. But this is small-scale noise; it doesn't affect the overall shape.

Iteration using `forEach` is an $O(n)$ operation, so its amortized plot should be a horizontal line. Its middle part indeed looks flat, but somewhere around 20,000 or so elements, it gradually slows down and starts to grow at a logarithmic rate. This is because even though `forEach` visits elements in increasing order, their locations in memory are essentially random: they were determined at the time the nodes were originally allocated for insertion. Red-black trees are terrible at keeping neighboring elements close to each other, and this makes them rather ill-suited for the memory architecture in today's computers.

But `forEach` is still much faster than `for-in`; incrementing an index is a relatively complicated $O(\log n)$ operation that essentially needs to look up an element from scratch, while the inorder walk always knows exactly where to find the next value.

The widening gap between `contains` and `for-in` is another demonstration of the effects of caching. The successive `index(after:)` calls in `for-in` visit new elements at random memory locations, but most of the path stays the same, so `for-in` is still relatively cache-friendly. Not so with `contains`, where every iteration visits a brand-new random path in the tree — memory accesses are rarely repeated, making the cache ineffective.

To see just how slow indexing-based iteration is in `RedBlackTree`, it's useful to compare it to our previous implementations. As you can see in figure 4.6, a `for-in` loop over the elements of a `RedBlackTree` is 200–1,000 times slower than over the elements of a `SortedArray`, and the curves are slowly diverging, proving the expected $O(\log n)$ vs. $O(1)$ algorithmic complexities.

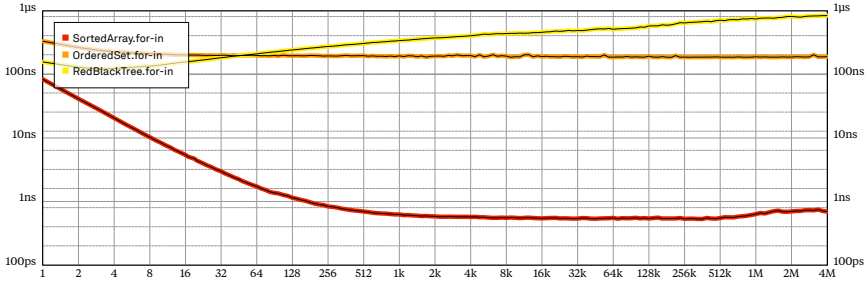



**Figure 4.6: Comparing the performance of three index(after:) implementations.**

We'll fix most of this slowdown in future chapters, but at least some of it is due to a (seemingly) unavoidable tradeoff between lookups and mutations: iteration is fastest when elements are stored in huge contiguous buffers, but such buffers are also the least convenient data structures for doing fast insertions. Any optimization that improves insertion performance is likely to involve breaking storage up into smaller chunks, which, as a side effect, makes iteration slower.

In red-black trees, elements are stored individually wrapped into separate heap-allocated tree nodes, which is probably the worst way to store elements if you want to quickly iterate over them. It doesn't help that in this particular design, we use an $O(\log n)$ algorithm to find the next index, but, perhaps surprisingly, this isn't where the `for` loop spends most of its time: replacing the `for-in` benchmark with `forEach` still leaves `RedBlackTree` 90–200 times slower than `SortedArray`.

Well, that's a bummer. But insertion gets faster, right? Well, let's see! Figure 4.7 compares the performance of `RedBlackTree.insert` to our previous implementations.

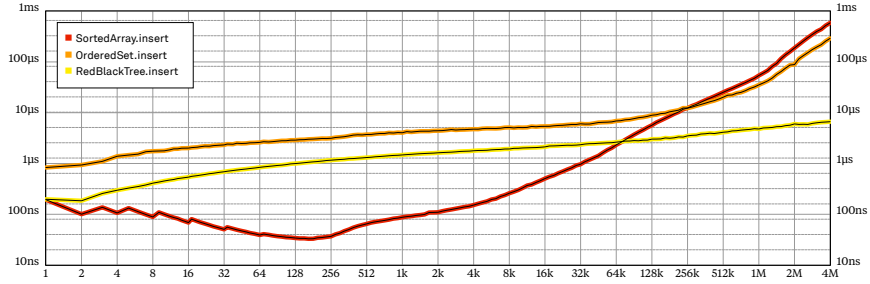



**Figure 4.7: Comparing the performance of three insert implementations.**

We can see that at large sizes, the algorithmic advantage of red-black trees clearly pays off. Inserting 4 million elements into a red-black tree is about 50–100 times faster than a sorted array. (It takes just 33 *seconds*, which compares nicely to `OrderedSet`'s 26 *minutes*.) If we added more elements, the gap would keep widening even further.

However, `SortedArray` clearly beats `RedBlackTree` when we have less than about 50,000 elements. Sorted arrays have much better locality of reference than red-black trees: even though we need to perform more memory accesses to insert a new element into a sorted array, these accesses are very close together. As long as we fit in the various caches (L1,

L2, TLB, what have you), the regularity of these accesses compensates for their larger number.

The growth rate of the `RedBlackTree.insert` curve is the best we can do — there's no data structure that solves the general `SortedSet` problem with better asymptotic performance. But this doesn't mean there's no room for improvement!

To make insertions faster, we need to start optimizing our implementation, speeding it up by some constant factor. This won't change the shape of its benchmark curve, but it will uniformly push it downward on the log-log chart.

Ideally, we want to push the `insert` curve down far enough so that it reaches or exceeds `SortedArray` performance for all sizes while maintaining the logarithmic growth rate.

# The Copy-On-Write Optimization

5

`RedBlackTree` creates a brand-new tree every time we insert a new element. The new tree shares some of its nodes with the original, but nodes on the entire path from the root to the new node are replaced with newly allocated nodes. This is the "easy" way to implement value semantics, but it's a bit wasteful.

When the nodes of a tree value aren't shared between other values, it's OK to mutate them directly — nobody will mind, because nobody else knows about that particular tree instance. Direct mutation will eliminate most of the copying and allocation, often resulting in a dramatic speedup.

We've already seen how Swift supports implementing optimized copy-on-write value semantics for reference types by providing the `isKnownUniquelyReferenced` function. Unfortunately, Swift 4 doesn't give us tools to implement COW for algebraic data types. We don't have access to the private reference type Swift uses to box up nodes, so we have no way to determine if a particular node is uniquely referenced. (The compiler itself doesn't have the smarts *(yet?)* to do the COW optimization on its own either.) There's also no easy way to directly access a value stored inside an enum case without extracting a separate copy of it. (Note that `Optional` does provide direct access to the value stored in it via its forced unwrapping `!` operator. However, the tools for implementing similar in-place access for our own enum types aren't available for use outside the standard library.)

So to implement COW, we need to give up on our beloved algebraic data types for now and rewrite everything into a more conventional (dare I say *tedious*?) imperative style, using boring old structs and classes, with a sprinkle of optionals.

# Basic Definitions

First, we need to define a public struct that'll represent sorted set values. The `RedBlackTree2` type below is just a small wrapper around a reference to a tree node that serves as its storage. `OrderedSet` worked the same way, so this pattern is already quite familiar to us by now:

```
public struct RedBlackTree2<Element: Comparable>: SortedSet {
  fileprivate var root: Node? = nil

  public init() {}
```

```
}
```

Next, let's take a look at the definition of our tree nodes:

```
extension RedBlackTree2 {
  class Node {
    var color: Color
    var value: Element
    var left: Node? = nil
    var right: Node? = nil
    var mutationCount: Int64 = 0

    init(_ color: Color, _ value: Element, _ left: Node?, _ right: Node?) {
      self.color = color
      self.value = value
      self.left = left
      self.right = right
    }
  }
}
```

Beyond the fields we had in the original enum case for `RedBlackTree.node`, this class also includes a new property called `mutationCount`. Its value is a count of how many times the node was mutated since its creation. This will be useful in our `Collection` implementation, where we'll use it to create a new design for indices. (We explicitly use a 64-bit integer so that it's unlikely the counter will ever overflow, even on 32-bit systems. Storing an 8-byte counter in every node isn't really necessary, as we'll only use the value stored inside root nodes. Let's ignore this fact for now; it's simpler to do it this way, and we'll find a way to make this less wasteful in the next chapter.)

But let's not jump ahead!

Using a separate type to represent nodes vs. trees means we can make the node type a private implementation detail to the public `RedBlackTree2`. External users of our collection won't be able to mess around with it, which is a nice bonus — everybody was able to see the internals of `RedBlackTree`, and anyone could use Swift's enum literal syntax to create any tree they wanted, which would easily break our red-black tree invariants.

The Node class is an implementation detail of the RedBlackTree2 struct; nesting the former in the latter expresses this relationship neatly. This also prevents naming conflicts with other types named Node in the same module. Additionally, it simplifies the syntax a little bit: Node automatically inherits RedBlackTree2's Element type parameter, so we don't have to explicitly specify it.

> Again, tradition dictates that the 1-bit color property should be packed into an unused bit in the binary representation of one of Node's reference properties; but it'd be unsafe, fragile, and fiddly to do that in Swift. It's better to simply keep color as a standalone stored property and to let the compiler set up storage for it.

Note that we essentially transformed the two-case RedBlackTree enum into optional references to the Node type. The .empty case is represented by a nil optional, while a non-nil value represents a .node. Node is a heap-allocated reference type, so we've made the previous solution's implicit boxing an explicit feature, giving us direct access to the heap reference and enabling the use of isKnownUniquelyReferenced.

# Rewriting Simple Lookup Methods

forEach is a nice example of how we can rewrite algorithms from algebraic data types to structs and classes. We typically need to make two methods — a public method for the wrapper struct, and a private method for the node type.

The struct method doesn't need to do anything if the tree is empty. Otherwise, it just needs to forward the call to the tree's root node. We can use optional chaining to express this succinctly:

```
extension RedBlackTree2 {
  public func forEach(_ body: (Element) throws -> Void) rethrows {
    try root?.forEach(body)
  }
}
```

The nodes' forEach method does most of the actual work for performing an inorder walk:

```
extension RedBlackTree2.Node {
  func forEach(_ body: (Element) throws -> Void) rethrows {
    try left?.forEach(body)
    try body(value)
    try right?.forEach(body)
  }
}
```

Optional chaining once again results in nice and compact code. Lovely!

We could write a similar recursive implementation of contains, but let's instead choose a more procedural solution. We'll use a node variable to navigate the tree, always moving it a step closer to the element we're looking for:

```
extension RedBlackTree2 {
  public func contains(_ element: Element) -> Bool {
    var node = root
    while let n = node {
      if n.value < element {
        node = n.right
      }
      else if n.value > element {
        node = n.left
      }
      else {
        return true
      }
    }
    return false
  }
}
```

This function distinctly resembles binary search from Chapter 2, even though it works with branching references rather than indices in a flat buffer. For our balanced trees, it also has the same logarithmic complexity.

# Tree Diagrams

Obviously we want to keep our cool tree diagrams, so let's rewrite the diagram method. The original was a method on RedBlackTree, and that got turned into an Optional<Node>. It's not currently possible to define extension methods on optional types wrapping generics, so the easiest way to rewrite the method is by turning it into a free-standing function:

```
private func diagram<Element>(for node: RedBlackTree2<Element>.Node?, _ top: String =
↪   "", _ root: String = "", _ bottom: String = "") -> String {
  guard let node = node else {
    return root + "•\n"
  }
  if node.left == nil && node.right == nil {
    return root + "\(node.color.symbol) \(node.value)\n"
  }
  return diagram(for: node.right, top + "    ", top + "┌──", top + "│   ")
    + root + "\(node.color.symbol) \(node.value)\n"
    + diagram(for: node.left, bottom + "│   ", bottom + "└──", bottom + "    ")
}

extension RedBlackTree2: CustomStringConvertible {
  public var description: String {
    return diagram(for: root)
  }
}
```

# Implementing Copy-On-Write

The easiest way to implement COW is to define little helper methods we can call to ensure we have unique references before mutating anything. We'll need to define one helper for each property we have that stores a reference value. In our case, that's three methods: one for the root reference inside the RedBlackTree2 struct, and one each for both child references (left and right) inside Node.

In all three cases, the helper method needs to check if its corresponding reference is unique, and if not, it needs to replace it with a new copy of its target. Making a copy of a node just means creating a new one with the same properties. Here's a simple function to do exactly that:

```
extension RedBlackTree2.Node {
  func clone() -> Self {
    return .init(color, value, left, right)
  }
}
```

Now let's define our COW helpers, starting with the one for the root node in RedBlackTree2:

```
extension RedBlackTree2 {
  fileprivate mutating func makeRootUnique() -> Node? {
    if root != nil, !isKnownUniquelyReferenced(&root) {
      root = root!.clone()
    }
    return root
  }
}
```

Remember our NSOrderedSet wrapper? It had a method called makeUnique that did something very similar. But this time, our reference is an optional, which makes things slightly more complicated. Thankfully, the standard library has an overload for isKnownUniquelyReferenced that takes an optional value, so we're covered on that front at least.

When we're doing low-level work in Swift, such as implementing COW or working with unsafe constructs, we often need to be much more careful about the precise semantics of our code. For COW, we need to know and care about the details of exactly how and when reference counts change, so that we can avoid creating temporary references that would break our uniqueness checks.

For example, if you're a seasoned Swift developer, you may be tempted to replace that clumsy-looking explicit nil check and the forced unwrapping with a healthy-looking conditional let binding, as in the snippet below:

```
if let root = self.root, !isKnownUniquelyReferenced(&root) { // BUG!
  self.root = root.clone()
}
```

This version may look marginally better, *but it doesn't do the same thing!* The `let` binding counts as a new reference to the root node, so in this code, `isKnownUniquelyReferenced` will always return `false`. This would completely break the COW optimization.

We're doing extremely fragile work here. If we get it wrong, our only indication will be that our code gets (much) slower — there's no runtime trap for making unnecessary copies!

On the other hand, if we make *fewer* copies than necessary, our code may sometimes mutate a shared reference type, which would break value semantics. That is a much more serious offense than a performance issue. If we're lucky, breaking value semantics will lead to runtime traps such as an unexpected invalid index. But usually the effects are subtler: the code that owns the references we ignored will sometimes produce incorrect results with no indication that anything went wrong. Good luck debugging that!

So be *very, very careful* about the code implementing COW. Even seemingly harmless cosmetic improvements may have extremely dire consequences.

Anyway, here are the other two helpers for the two child references inside a node. They're both straightforward adaptations of `makeRootUnique`:

```
extension RedBlackTree2.Node {
  func makeLeftUnique() -> RedBlackTree2<Element>.Node? {
    if left != nil, !isKnownUniquelyReferenced(&left) {
      left = left!.clone()
    }
    return left
  }

  func makeRightUnique() -> RedBlackTree2<Element>.Node? {
    if right != nil, !isKnownUniquelyReferenced(&right) {
      right = right!.clone()
    }
    return right
```

```
  }
}
```

# Insertion

Now we're ready to start rewriting insert. Again, we need to split it into two parts: a public method in the wrapper struct, and a private one in the node class. (insert was already split into smaller parts doing individual subtasks, so this is mainly a matter of deciding what goes where.)

The wrapper method is the easier of the two. It's a mutating method, so it must call makeRootUnique to ensure it's OK to mutate the root node. If the tree is empty, we need to create a new root node for the inserted element; otherwise we forward the insertion call to the existing root node, which is now guaranteed to be unique:

```
extension RedBlackTree2 {
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪ Element) {
    guard let root = makeRootUnique() else {
      self.root = Node(.black, element, nil, nil)
      return (true, element)
    }
    defer { root.color = .black }
    return root.insert(element)
  }
}
```

We use a defer statement to enforce the first red-black requirement, unconditionally painting the root black.

The node's insert method corresponds to our old friend _inserting in RedBlackTree:

```
extension RedBlackTree2.Node {
  func insert(_ element: Element)  -> (inserted: Bool, memberAfterInsert: Element) {
    mutationCount += 1
```

```
      if element < self.value {
        if let next = makeLeftUnique() {
          let result = next.insert(element)
          if result.inserted { self.balance() }
          return result
        }
        else {
          self.left = .init(.red, element, nil, nil)
          return (inserted: true, memberAfterInsert: element)
        }
      }
      if element > self.value {
        if let next = makeRightUnique() {
          let result = next.insert(element)
          if result.inserted { self.balance() }
          return result
        }
        else {
          self.right = .init(.red, element, nil, nil)
          return (inserted: true, memberAfterInsert: element)
        }
      }
      return (inserted: false, memberAfterInsert: self.value)
    }
  }
```

Annoyingly, we had to convert the switch statement into an if cascade and add a bunch of assignments for mutation. But it still has the same structure as the original code — in fact, I can *almost* imagine the Swift compiler doing this rewrite automatically. (Algebraic data types with automagical COW optimization would be a nice compiler improvement for, say, perhaps Swift 10 or so.)

But now we get to balancing. Oh man.

RedBlackTree had a memorable, graceful balancing implementation based on pattern matching — *an elegant weapon for a more civilized age*. Such elegance. Very beauty!

And now we need to go and smear a bunch of stinking assignment statements all over it:

```
extension RedBlackTree2.Node {
  func balance() {
    if self.color == .red  { return }
    if left?.color == .red {
      if left?.left?.color == .red {
        let l = left!
        let ll = l.left!
        swap(&self.value, &l.value)
        (self.left, l.left, l.right, self.right) = (ll, l.right, self.right, l)
        self.color = .red
        l.color = .black
        ll.color = .black
        return
      }
      if left?.right?.color == .red {
        let l = left!
        let lr = l.right!
        swap(&self.value, &lr.value)
        (l.right, lr.left, lr.right, self.right) = (lr.left, lr.right, self.right, lr)
        self.color = .red
        l.color = .black
        lr.color = .black
        return
      }
    }
    if right?.color == .red {
      if right?.left?.color == .red {
        let r = right!
        let rl = r.left!
        swap(&self.value, &rl.value)
        (self.left, rl.left, rl.right, r.left) = (rl, self.left, rl.left, rl.right)
        self.color = .red
        r.color = .black
        rl.color = .black
        return
      }
      if right?.right?.color == .red {
        let r = right!
        let rr = r.right!
```

```
            swap(&self.value, &r.value)
            (self.left, r.left, r.right, self.right) = (r, self.left, r.left, rr)
            self.color = .red
            r.color = .black
            rr.color = .black
            return
          }
        }
      }
    }
```

Yes, this is really what our original balancing function turns into after what amounts to a straightforward rewrite to an imperative style. It breaks my heart to see it.

We could make this code a little bit easier on the eyes (and probably faster) by refactoring it. For example, we could inline `balance` into `insert`, eliminating some of the branches by taking into account information about which direction we inserted the new element.

But it'd likely be easier to just open a book on data structures and port the red-black insertion algorithm we find there into Swift. We aren't going to do that here though, because it isn't very interesting, and it wouldn't actually speed things up by much.

> If you're interested though, check out the red-black tree implementation I made for Swift 2. I haven't updated it for Swift 3 or 4 because there was no point in doing so — it turned out B-trees, which we're going to discuss in the following chapters, work much better.)

# Implementing Collection

Let's see if this time we can make a better index implementation.

For `RedBlackTree`, we used a dummy index type that was just a wrapper around an element value. To implement `index(after:)`, we needed to find the element in the tree from scratch, which is an operation with logarithmic complexity.

This means that when we use indexing to iterate over an entire RedBlackTree, we're doing it in $O(n \log n)$ steps, where $n$ is the count of the collection. That's no good at all — iterating over all elements in a collection should only take $O(n)$ time! (This isn't an actual Collection requirement, but it's definitely a reasonable expectation.)

Well. What can we do? One way to speed things up is to have our indices include the entire path from the root to a particular element.

However, a wrinkle in this idea is that indices aren't supposed to hold strong references to parts of the collection. So let's represent the path with an array of zeroing weak references instead.

## Index Definition

Swift 4 doesn't allow us to directly define an array of weak references, so first we need to define a simple wrapper struct holding a weak reference:

```
private struct Weak<Wrapped: AnyObject> {
  weak var value: Wrapped?

  init(_ value: Wrapped) {
    self.value = value
  }
}
```

We can now declare the path as an array of these Weak values and it'll work fine. We'll have to remember to append .value in the source to extract the actual reference (as in path[0].value), but this is just a cosmetic issue; the wrapper struct has no runtime penalty whatsoever. It's a little bit annoying to have to do this dance; adding language support for weak/unowned arrays seems like a nice candidate for a *swift-evolution* proposal. But we're here to implement a sorted set and not to enhance Swift itself, so let's accept this little language blemish for the time being and move on.

Now we're ready to define the actual index type. It's basically a wrapper around a path, which is an array of weakly held nodes. To make it easy to validate indices, we also include a direct weak reference to the root node and a copy of the root node's mutation count at the time the index was created:

```
extension RedBlackTree2 {
  public struct Index {
    fileprivate weak var root: Node?
    fileprivate let mutationCount: Int64?

    fileprivate var path: [Weak<Node>]

    fileprivate init(root: Node?, path: [Weak<Node>]) {
      self.root = root
      self.mutationCount = root?.mutationCount
      self.path = path
    }
  }
}
```

Figure 5.1 depicts an example index instance, pointing to the first value in a simple red-black tree. Note how the index uses weak references so that it doesn't retain any of the tree nodes. The isKnownUniquelyReferenced function only counts strong references, so this allows in-place mutations when there are outstanding indices. However, this implies that mutations may break node references inside previously created indices, so we'll have to be much more careful about index validation.

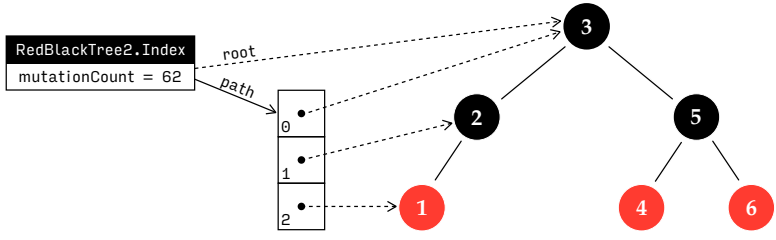



**Figure 5.1: The start index of a simple red-black tree, as implemented by RedBlackTree2.Index. Dashed lines represent weak references.**

## Start and End Indices

Let's start writing Collection methods.

We can define the endIndex as an index with an empty path, which is easy:

```
extension RedBlackTree2 {
  public var endIndex: Index {
    return Index(root: root, path: [])
  }
```

The start index isn't too difficult either — we just have to construct a path to the leftmost element inside the tree:

```
  // - Complexity: O(log(n)); this violates `Collection` requirements.
  public var startIndex: Index {
    var path: [Weak<Node>] = []
    var node = root
    while let n = node {
      path.append(Weak(n))
      node = n.left
    }
    return Index(root: root, path: path)
  }
}
```

This takes $O(\log n)$ time, once again breaking Collection's corresponding requirement. It would certainly be possible to fix this, but it seems trickier than in RedBlackTree, and it's unclear if the benefits would outweigh the effort, so we'll leave it like this for now. If this were a real package, we'd need to add a warning to the documentation pointing out the issue, in order to let our users know about it. (Some collection operations may become a lot slower as a result of this.)

## Index Validation

Let's discuss index validation next. For `RedBlackTree2`, we need to invalidate all existing indices after every mutation, because the mutation may have modified or replaced some of the nodes on the path stored inside them.

To do that, we need to verify that the tree's root is identical to the root reference in the index, and that they have the same mutation count. The root may be `nil` for a valid index in an empty tree, which complicates matters a little:

```
extension RedBlackTree2.Index {
  fileprivate func isValid(for root: RedBlackTree2<Element>.Node?) -> Bool {
    return self.root === root
      && self.mutationCount == root?.mutationCount
  }
}
```

If this function doesn't trap, then we know it belongs to the correct tree, and also that the tree hasn't been mutated since the index was created. It's then safe to assume the index path consists of valid references. Otherwise, the path may contain zeroed references or nodes with modified values, so we must not look at it.

To perform index comparisons, we need to check that the two indices belong to the same tree and that they're both valid for that tree. Here's a small static function to do that:

```
extension RedBlackTree2.Index {
  fileprivate static func validate(_ left: RedBlackTree2<Element>.Index, _ right:
  ↪ RedBlackTree2<Element>.Index) -> Bool {
    return left.root === right.root
      && left.mutationCount == right.mutationCount
      && left.mutationCount == left.root?.mutationCount
  }
}
```

## Subscripting

Subscripting by an index has a really exciting implementation because it contains not one, but two exclamation marks:

```
extension RedBlackTree2 {
  public subscript(_ index: Index) -> Element {
    precondition(index.isValid(for: root))
    return index.path.last!.value!.value
  }
}
```

The use of force unwraps here is justified — subscripting with `endIndex` should trap, and so should subscripting with an invalidated index. We could provide better error messages, but this will do just fine for now.

## Index Comparison

Indices must be comparable. To implement this, we can simply retrieve the nodes at the end of the two index paths and compare their values, like we did in `RedBlackTree`. Let's see how we can get the current node of an index:

```
extension RedBlackTree2.Index {
  /// Precondition: `self` is a valid index.
  fileprivate var current: RedBlackTree2<Element>.Node? {
    guard let ref = path.last else { return nil }
    return ref.value!
  }
}
```

This property assumes the index is valid; otherwise, the force unwrap may fail and trap, crashing our program.

We can now write the code for doing comparisons; it's similar to what we had before:

```
extension RedBlackTree2.Index: Comparable {
```

```
  public static func ==(left: RedBlackTree2<Element>.Index, right:
  ↪ RedBlackTree2<Element>.Index) -> Bool {
    precondition(RedBlackTree2<Element>.Index.validate(left, right))
    return left.current === right.current
  }

  public static func <(left: RedBlackTree2<Element>.Index, right:
  ↪ RedBlackTree2<Element>.Index) -> Bool {
    precondition(RedBlackTree2<Element>.Index.validate(left, right))
    switch (left.current, right.current) {
    case let (a?, b?):
      return a.value < b.value
    case (nil, _):
      return false
    default:
      return true
    }
  }
}
```

## Index Advancement

Finally, we need to implement index advancement. We'll do this by forwarding the job to a mutating method on the index type, which we'll write in a moment:

```
extension RedBlackTree2 {
  public func formIndex(after index: inout Index) {
    precondition(index.isValid(for: root))
    index.formSuccessor()
  }

  public func index(after index: Index) -> Index {
    var result = index
    self.formIndex(after: &result)
    return result
  }
}
```

To find the next index after an existing one, we need to look for the leftmost value under the current node's right subtree. If the current node has no right child, we need to back up to the nearest ancestor that has the current node inside its *left* subtree.

If there's no such ancestor node, then we've reached the end of the collection. Having an empty path makes the index equal to the collection's endIndex in such a case, which is exactly what we want:

```
extension RedBlackTree2.Index {
  /// Precondition: `self` is a valid index other than `endIndex`.
  mutating func formSuccessor() {
    guard let node = current else { preconditionFailure() }
    if var n = node.right {
      path.append(Weak(n))
      while let next = n.left {
        path.append(Weak(next))
        n = next
      }
    }
    else {
      path.removeLast()
      var n = node
      while let parent = self.current {
        if parent.left === n { return }
        n = parent
        path.removeLast()
      }
    }
  }
}
```

For BidirectionalCollection, we also need to be able to step backward from any index. The code for doing this has the exact same structure as index(before:), except we need to swap left and right, and we need to handle the endIndex specially:

```
extension RedBlackTree2 {
  public func formIndex(before index: inout Index) {
    precondition(index.isValid(for: root))
```

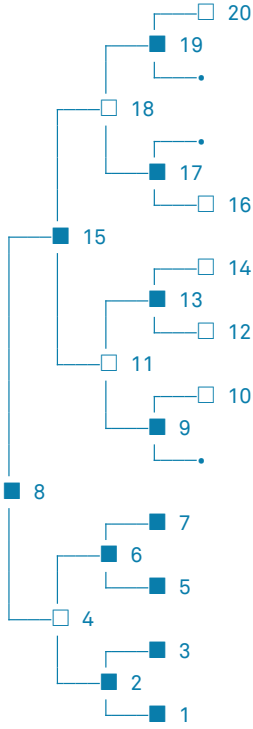
```
    index.formPredecessor()
  }

  public func index(before index: Index) -> Index {
    var result = index
    self.formIndex(before: &result)
    return result
  }
}

extension RedBlackTree2.Index {
  /// Precondition: `self` is a valid index other than `startIndex`.
  mutating func formPredecessor() {
    let current = self.current
    precondition(current != nil || root != nil)
    if var n = (current == nil ? root : current!.left) {
      path.append(Weak(n))
      while let next = n.right {
        path.append(Weak(next))
        n = next
      }
    }
    else {
      path.removeLast()
      var n = current
      while let parent = self.current {
        if parent.right === n { return }
        n = parent
        path.removeLast()
      }
    }
  }
}
```

# Examples

Phew, we're all done! Let's play around with this new type here:

```
var set = RedBlackTree2<Int>()
for i in (1 ... 20).shuffled() {
  set.insert(i)
}
```

▶ set




▶ set.contains(13)

true

▶ set.contains(42)

false

▶ set.filter { $0 % 2 == 0 }

[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

# Performance Benchmarks

Running our usual benchmarks on RedBlackTree2 gets us the results in figure 5.2. The curves for insert, contains, and forEach have shapes that are roughly similar to what we saw with RedBlackTree. But the for-in benchmark looks really weird!

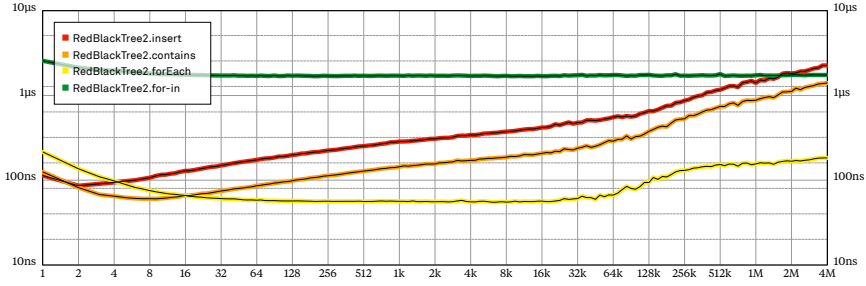



**Figure 5.2: Benchmark results for RedBlackTree2 operations. The plot shows input size vs. average execution time for a single iteration on a log-log chart.**

## Optimizing Iteration Performance

We redesigned our index type so that index(after:) has amortized constant time. We seem to have succeeded at this: the for-in curve is nicely flat, with only the slightest increase at large set sizes. But it's way high up on the chart! For sets smaller than 100,000 elements, apparently it's faster to insert an element than it is to iterate over one. That makes no sense.

Figure 5.3 compares for-in performance between RedBlackTree and RedBlackTree2. We can see that RedBlackTree2.index(after:) is indeed an (amortized) constant-time operation: its curve remains flat even for huge sets. RedBlackTree's logarithmic index advancement is asymptotically worse, but it starts much lower, so it only gets slower than RedBlackTree2 at about 16 million elements, which is far beyond the range displayed on our benchmarking chart. This isn't very impressive at all.

To make RedBlackTree2 competitive with the primitive indices of RedBlackTree, we need to speed up its indexing operations by a factor of four. While this should be possible, it's not immediately obvious how we can achieve it. Regardless, let's try!

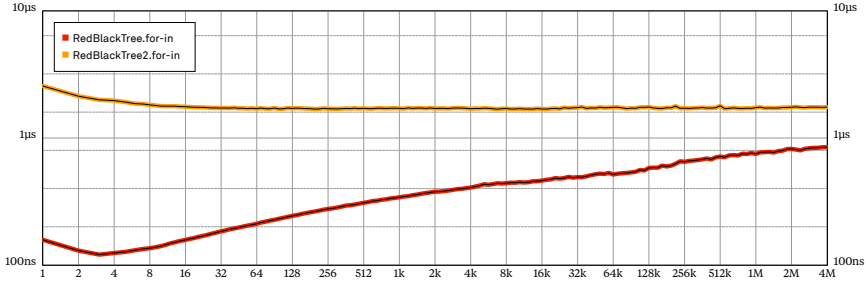



**Figure 5.3: Comparing the performance of for-in implementations in tree-based sorted sets.**

One idea is to eliminate index validation, which is a major contributor to the cost of iteration. Completely removing it would normally be a bad idea, but in the special case of a for-in loop, it's unnecessary: the collection remains the same throughout the iteration, so validation can never fail. To get rid of it, we can replace Collection's default IndexingIterator with a specialized implementation of IteratorProtocol that handles indices directly:

```
extension RedBlackTree2 {
  public struct Iterator: IteratorProtocol {
    let tree: RedBlackTree2
    var index: RedBlackTree2.Index

    init(_ tree: RedBlackTree2) {
      self.tree = tree
      self.index = tree.startIndex
    }

    public mutating func next() -> Element? {
      if index.path.isEmpty { return nil }
      defer { index.formSuccessor() }
      return index.path.last!.value!.value
    }
  }
}

extension RedBlackTree2 {
  public func makeIterator() -> Iterator {
```

```
        return Iterator(self)
    }
}
```

Note how the iterator includes an apparently redundant copy of the tree. Even though we never explicitly use this stored property in `next`, it still serves a critically important purpose: it retains the root node for us, guaranteeing that the tree doesn't get deallocated during the lifetime of the iterator.

Defining this custom iterator speeds up our `for-in` benchmark by a factor of two. (See the `for-in2` curve in figure 5.4.) That's a nice speedup, but we wanted a 400% improvement, so let's try to find some other way to speed iteration up even further.

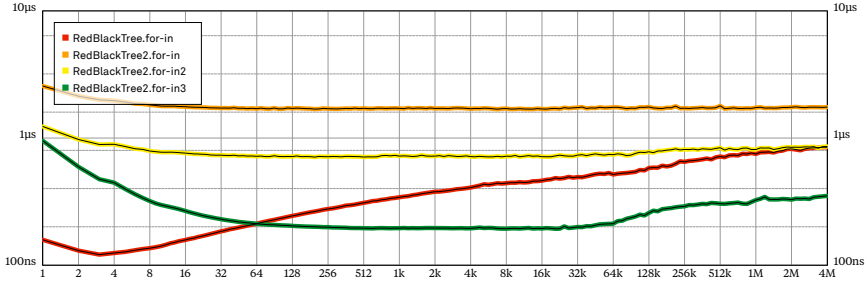



**Figure 5.4: Comparing the performance of for-in implementations in tree-based sorted sets. RedBlackTree2.for-in2 eliminates unnecessary index validation steps by implementing a custom iterator, while for-in3 replaces zeroing weak references with unowned-unsafe ones.**

A key observation is that our mutation count checks guarantee that the path of a valid index never has any expired weak references — because the tree it's associated with is still around, with the exact same nodes it had at the time the index was created. This implies that we can safely replace the path's weak references with `unowned(unsafe)` ones. Doing so eliminates all reference count management operations, speeding up iteration by another factor of two. (See the `for-in3` curve on the same figure.) This gets us to acceptable `for-in` performance, so we can stop optimizing for now.

Choosing an asymptotically better indexing algorithm in `RedBlackTree2` didn't automatically provide faster results: the algorithmic advantage was swamped by less efficient implementation details. However, we were able to make the "better" algorithm competitive with the "worse" one by optimizing it. In this case, we got lucky, and we

reached adequate performance after just two relatively simple optimization steps. Often we need to do much more!

## Insertion Performance

With RedBlackTree2, we now have a really efficient, tree-based SortedSet implementation. As figure 5.5 demonstrates, implementing in-place mutations resulted in a 300–500% boost in insertion performance. RedBlackTree2.insert is no slouch; it completes 4 million insertions in just 12 seconds.

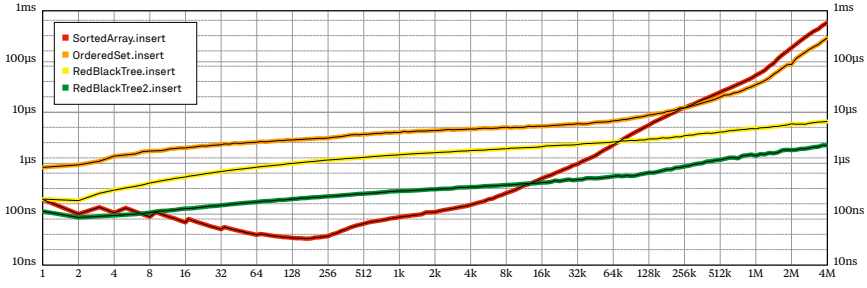



**Figure 5.5: Comparing the performance of four insert implementations.**

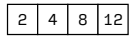
But it's still unable to outperform SortedArray below 8,000 elements: the latter can be as much as four times faster. Hmm. While it's certainly possible to optimize red-black tree insertions even further, it seems unlikely we'll be able to achieve a 400% speedup. What next, then?

# B-Trees

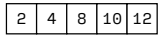
6

The raw performance of a small sorted array seems impossible to beat. So let's not even try; instead, let's build a sorted set entirely out of small sorted arrays!

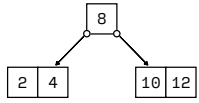
It's easy enough to get started: we just insert elements into a single array until we reach some predefined maximum size. Let's say we want to keep our arrays below five elements and we've already inserted four values:




If we now need to insert the value 10, we get an array that's too large:

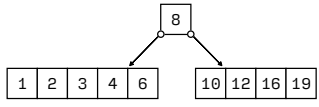



We can't keep it like this, so we have to do something. One option is to cut the array into two halves, extracting the middle element to serve as a sort of separator value between them:

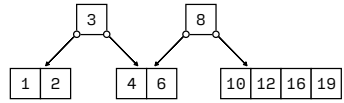



Now we have a cute little tree structure with leaf nodes containing small sorted arrays. This seems like a promising approach: it combines sorted arrays and search trees into a single unified data structure, hopefully giving us array-like super-fast insertions at small sizes, all while retaining tree-like logarithmic performance for large sets. Synergy!

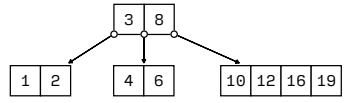
Let's see what happens when we try adding more elements. We can keep inserting new values in the two leaf arrays until one of them gets too large again:




When this happens, we need to do another split, giving us three arrays and two separators:

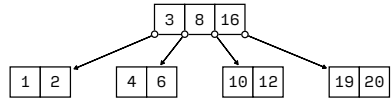



What should we do with those two separator values? We store all other elements in sorted arrays, so it seems like a reasonable idea to put these separators in a sorted array of their own:

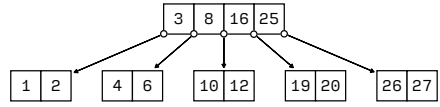



So apparently each node in our new array-tree hybrid will hold a small sorted array. Nice and consistent; I love it so far.

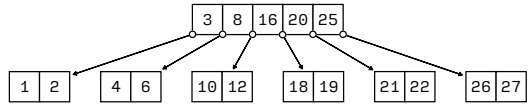
Now let's say we want to insert the value 20 next. This goes in the rightmost array, which already has four elements — so we need to do another split, extracting 16 from the middle as a new separator. No biggie, we can just insert it into the array at the top:



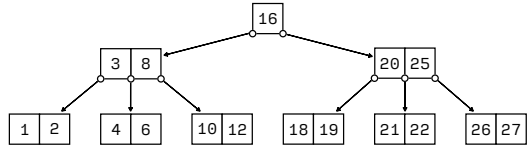

Lovely! We can keep going like this; inserting 25, 26, and 27 overflows the rightmost array again, extracting the new separator, 25:



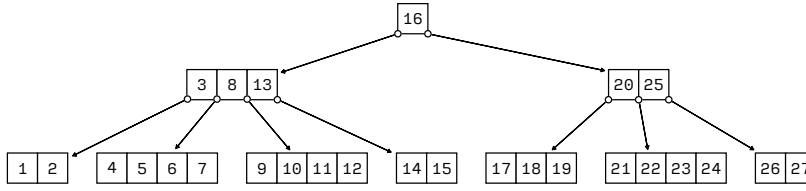

However, the top array is now full, which is worrying. Let's say we insert 18, 21, and 22 next, so we end up with the situation below:



What next? We can't leave the separator array bloated like this. Previously, we handled oversized arrays by splitting them — why not try doing the same now?




Oh, how neat: splitting the second-level array just adds a third level to our tree. This gives us a way to keep inserting new elements indefinitely; when the third-level array fills up, we can just add a fourth level, and so on, ad infinitum:




We've just invented a brand-new data structure! This is so exciting.

Unfortunately, it turns out that Rudolf Bayer and Ed McCreight already had the same idea way back in 1971, and they named their invention *B-tree*. What a bummer! I wrote an entire book to introduce a thing that's almost half a century old. How embarrassing.

Fun fact: red-black trees were actually derived from a special case of B-trees in 1978. These are all ancient data structures.

# B-Tree Properties

As we've seen, *B-trees* are search trees like red-black trees, but their layout is different. In a B-tree, nodes may have hundreds (or even thousands) of children, not just two. The number of children isn't entirely unrestricted, though: this number must fit inside a certain range.

The maximum number of children a node may have is determined when the tree is created — this number is called the *order* of a B-tree. (The order of a B-tree has nothing to do with the ordering of its elements.) Note that the maximum number of elements stored in a node is one less than the tree's order. This can be a source of off-by-one indexing errors, so it's important to keep in mind as we work on B-tree internals.

In the previous section, we built an example B-tree of order 5. In practice, the order is typically between 100 and 2,000, so 5 is unusually low. However, nodes with 1,000 children wouldn't easily fit on the page; it's easier to understand B-trees by drawing toy examples.

To help in finding our way inside the tree, each internal node stores one element between each of its children, similar to how a node in a red-black tree contains a single value sorted between its left and right subtrees. So a B-tree node with 1,000 children will contain 999 elements neatly sorted in increasing order.

To keep lookup operations fast, B-trees maintain the following balancing invariants:

1. **Maximum size:** All nodes store a maximum of `order - 1` elements, sorted in increasing order.
2. **Minimum size:** Non-root nodes must never be less than half full, i.e. the number of elements in all nodes except the root node must be at least `(order - 1) / 2`.
3. **Uniform depth:** All leaf nodes must be at the same depth in the tree, counting from the root node at the top.

Note that the last two properties are a natural consequence of the way we do insertions; we didn't have to do anything special to prevent nodes from getting too small or to make sure all leaves are on the same level. (These properties are much harder to maintain in other B-tree operations, though. For instance, removing an element may result in an undersized node that needs to be fixed.)

According to these rules, nodes in a B-tree of order 1,000 will hold anywhere between 499 and 999 elements (inclusive). The single exception is the root node, which isn't constrained by the lower bound: it may contain 0 to 999 elements. (This is so we can create B-trees with, say, less than 499 elements.) Therefore, a single B-tree node on its own may have the same number of elements as a red-black tree that is *10–20 levels* deep!

Storing so many elements in a single node has two major advantages:

1. **Reduced memory overhead.** Each value in a red-black tree is stored in a separate, heap-allocated node that also includes a pointer to a method lookup table, its reference count, and the two references for the node's left and right children. Nodes in a B-tree store elements in bulk, which can considerably reduce memory allocation overhead. (The exact savings depends on the size of the elements and the order of the tree.)
2. **Access patterns better suited for memory caching.** B-tree nodes store their elements in small contiguous buffers. If the buffers are sized so that they fit well within the L1 cache of the CPU, operations working on them can be considerably faster than the equivalent code operating on red-black trees, the values of which are scattered all over the heap.

To understand how dense B-trees really are, consider that adding an extra level to a B-tree multiplies the maximum number of elements it may store by its order. For example, here's how the minimum and maximum element counts grow as we increase the depth of a B-tree of order 1,000:

| Depth | Minimum size | Maximum size |
|---:|---:|---:|
| 1 | 0 | 999 |
| 2 | 999 | 999 999 |
| 3 | 499 999 | 999 999 999 |
| 4 | 249 999 999 | 999 999 999 999 |
| 5 | 124 999 999 999 | 999 999 999 999 999 |
| 6 | 62 499 999 999 999 | 999 999 999 999 999 999 |
| ⋮ | ⋮ | ⋮ |
| $n$ | $2\left\lceil \frac{order}{2} \right\rceil^{n-1} - 1$ | $order^n - 1$ |

Clearly, we'll rarely, if ever, work with B-trees that are more than a handful of levels deep. Theoretically, the depth of a B-tree is $O(\log n)$, where $n$ is the number of elements. But the base of this logarithm is so large that, given the limited amount of memory available in any real-world computer, it's in fact not too much of a stretch to say that B-trees have $O(1)$ depth for any input size we may reasonably expect to see in practice.

> This last sentence somehow seems to make sense, even though the same line of thinking would lead me to say that any practical algorithm runs in $O(1)$ time, with the constant limit being my remaining lifetime — obviously, I don't find algorithms that finish after my death very practical at all. On the other hand, you could arrange all particles in the observable universe into a 1,000-order B-tree that's just 30 levels or so deep; so it does seem rather pointless and nitpicky to keep track of the logarithm.

Another interesting consequence of such a large fan-out is that in B-trees, the *vast* majority of elements are stored in leaf nodes. Indeed, in a B-tree of order 1,000, at least 99.8% of elements will always reside in leaves. Therefore, for operations that process B-tree elements in bulk (such as iteration), we mostly need to concentrate our optimization efforts on making sure leaf nodes are processed fast: the time spent on internal nodes often won't even register in benchmarking results.

Weirdly, despite this, the number of nodes in a B-tree is still technically proportional to the number of its elements, and most B-tree algorithms have the same "big-O" performance as the corresponding binary tree code. In a way, B-trees play tricks with the simplifying assumptions behind the big-O notation: constant factors often do matter in practice. This should be no surprise by now. After all, if we didn't care about constant factors, we could've ended this book after the chapter on RedBlackTree!

# Basic Definitions

Enough talking, let's get to work!

We begin implementing B-trees by defining a public wrapper struct around a root node reference, just like we did in RedblackTree2:

```
public struct BTree<Element: Comparable> {
  fileprivate var root: Node

  init(order: Int) {
    self.root = Node(order: order)
  }
}
```

In RedblackTree2, root was an optional reference so that empty trees didn't need to allocate anything. Making the root non-optional will make our code a little bit shorter though, and this time brevity makes sense.

The node class needs to hold two buffers: one for storing elements, and the other for storing child references. The simplest approach is to use arrays, so let's start with that.

To make it easier to test our tree, the tree order is not a compile-time constant: we're able to customize it using the initializer above. To accommodate this, we'll store the order inside every node as a read-only stored property, for easy access:

```
extension BTree {
  final class Node {
    let order: Int
    var mutationCount: Int64 = 0
    var elements: [Element] = []
    var children: [Node] = []

    init(order: Int) {
      self.order = order
    }
  }
}
```

We also added the mutationCount property from the previous chapter. As promised, storing a per-node mutation count is now a lot less wasteful — a typical node will contain several kilobytes of data, so adding an extra 8 bytes won't matter at all:

Figure 6.1 shows how our example B-tree would be represented as a BTree. Note the arrangement of indices in the elements and children arrays: for all i in 0 ..< elements.count, the value elements[i] is larger than any value in children[i], but less than all values in children[i + 1].

# The Default Initializer

It's nice that we allow the user to set a custom tree order, but we also need to define a parameterless initializer for SortedSet. What order should we use by default?

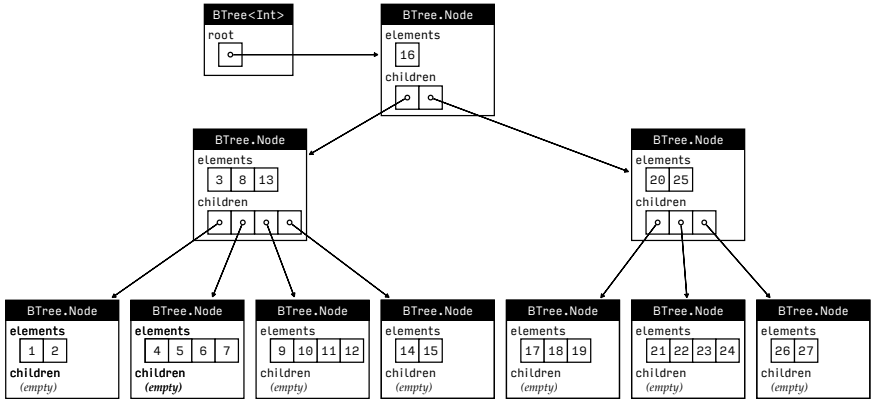



**Figure 6.1: The B-tree on page 102, as represented by BTree.**

One option is to just set a concrete default in the code:

```
extension BTree {
  public init() { self.init(order: 1024) }
}
```

However, we can do better than that. Benchmarking indicates that B-trees are fastest when the maximum size of the element buffer is about a quarter of the size of the L1 memory cache inside the CPU. Darwin OS (the kernel underlying all of Apple's operating systems) provides a sysctl query for getting the size of the cache, while Linux has a sysconf query for the same. Here's one way to access these from Swift:

```
#if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
import Darwin
#elseif os(Linux)
import Glibc
#endif

public let cacheSize: Int? = {
  #if os(macOS) || os(iOS) || os(watchOS) || os(tvOS)
    var result: Int = 0
    var size = MemoryLayout<Int>.size
```

```
    let status = sysctlbyname("hw.l1dcachesize", &result, &size, nil, 0)
    guard status != -1 else { return nil }
    return result
  #elseif os(Linux)
    let result = sysconf(Int32(_SC_LEVEL1_DCACHE_SIZE))
    guard result != -1 else { return nil }
    return result
  #else
    return nil // Unknown platform
  #endif
}()
```

> There are lots of other sysctl names on Darwin; you can get a list of the most popular and useful ones by typing man 3 sysctl in a Terminal window. Also, sysctl -a gets you a full list of every available query along with their current values. On Linux, getconf -a gets you a similar list, and the confnames.h header file lists the symbolic names you can use as a parameter for sysconf.

Once we know the cache size, we can use it to define the no-params initializer using the standard library's MemoryLayout construct to find the number of bytes an element needs inside a contiguous memory buffer:

```
extension BTree {
  public init() {
    let order = (cacheSize ?? 32768) / (4 * MemoryLayout<Element>.stride)
    self.init(order: Swift.max(16, order))
  }
}
```

If we can't determine the cache size, we use a reasonable-looking default value. The call to max ensures that we'll have an adequately bushy tree even if the elements are huge.

# Iteration with forEach

Let's see what forEach looks like in a B-tree. Like before, the wrapper struct's forEach method simply forwards the call to the root node:

```
extension BTree {
  public func forEach(_ body: (Element) throws -> Void) rethrows {
    try root.forEach(body)
  }
}
```

But how do we visit all elements under a node? Well, if the node is a leaf node, we just have to call body on all its elements. If the node has children, we need to interleave element visits with recursive calls to visit each of the child nodes, one by one:

```
extension BTree.Node {
  func forEach(_ body: (Element) throws -> Void) rethrows {
    if children.isEmpty {
      try elements.forEach(body)
    }
    else {
      for i in 0 ..< elements.count {
        try children[i].forEach(body)
        try body(elements[i])
      }
      try children[elements.count].forEach(body)
    }
  }
}
```

Note that elements under the first child have values that are less than the first element, so we need to start with a recursive call. Also, there's an extra child beyond the last element which contains elements that are larger than it is — we have to add an additional call to cover it as well.

# Lookup Methods

In order to find a specific element in a B-tree, we first need to write a utility method for finding an element inside a particular node.

Since the elements inside a node are stored in a sorted array, this task is reduced to the same binary search algorithm we wrote for SortedArray. To make things easier, this version of the function also incorporates the final membership test:

```
extension BTree.Node {
  internal func slot(of element: Element) -> (match: Bool, index: Int) {
    var start = 0
    var end = elements.count
    while start < end {
      let mid = start + (end - start) / 2
      if elements[mid] < element {
        start = mid + 1
      }
      else {
        end = mid
      }
    }
    let match = start < elements.count && elements[start] == element
    return (match, start)
  }
}
```

We're going to call indices inside a node *slots*, in order to differentiate them from the collection indices we'll define later. (A slot may be used as an index in either array — elements or children — inside the node.)

We now have a way to calculate slots, which is exactly what we need to write an implementation for contains. Once again, the wrapper struct just forwards the call to the root node:

```
extension BTree {
  public func contains(_ element: Element) -> Bool {
```

```
    return root.contains(element)
  }
}
```

Node.contains starts by calling slot(of:). If it finds a matching slot, then the element we're looking for is definitely in the tree, so contains should return true. Otherwise, we can use the returned index value to narrow the search to a specific child node:

```
extension BTree.Node {
  func contains(_ element: Element) -> Bool {
    let slot = self.slot(of: element)
    if slot.match { return true }
    guard !children.isEmpty else { return false }
    return children[slot.index].contains(element)
  }
}
```

As we can see, B-trees combine sorted array algorithms and search tree algorithms into something new and exciting. (Given their age, calling B-trees *new* might be a stretch, but they're definitely exciting, aren't they?)

# Implementing Copy-On-Write

The first cut of COW for B-trees will use the same sort of helper methods we've written like a billion times by now. The helper for the root node is especially boring:

```
extension BTree {
  fileprivate mutating func makeRootUnique() -> Node {
    if isKnownUniquelyReferenced(&root) { return root }
    root = root.clone()
    return root
  }
}
```

Yawn!

The clone method needs to do a shallow copy of the node properties. Note that elements and children are arrays, so they have their own implementation for COW. This is somewhat redundant in this case, but it does make our code shorter:

```
extension BTree.Node {
  func clone() -> BTree<Element>.Node {
    let clone = BTree<Element>.Node(order: order)
    clone.elements = self.elements
    clone.children = self.children
    return clone
  }
}
```

We don't need to copy the mutation counter, because we'll never compare its value between different nodes: it's only used to differentiate between versions of one particular node instance.

Children are stored in an array rather than as individual properties, so their COW helper is slightly more interesting because we need to add a parameter that tells the function which child we intend to mutate later:

```
extension BTree.Node {
  func makeChildUnique(at slot: Int) -> BTree<Element>.Node {
    guard !isKnownUniquelyReferenced(&children[slot]) else {
      return children[slot]
    }
    let clone = children[slot].clone()
    children[slot] = clone
    return clone
  }
}
```

Luckily, Swift arrays include super-secret special magic sauce so that their subscripts allow in-place mutation of their elements. The same magic enables isKnownUniquelyReferenced to keep working even though we're calling it on a subscript expression and not a simple stored property. (Unfortunately, it's not yet possible to provide these so-called *mutating addressors* in our own collections; the technology behind them is immature and is only available to the standard library.)

# Utility Predicates

Before we move on, it's worth defining a couple of predicate properties on node values:

```
extension BTree.Node {
  var isLeaf: Bool { return children.isEmpty }
  var isTooLarge: Bool { return elements.count >= order }
}
```

We'll use the `isLeaf` property to determine if an instance is a leaf node, while `isTooLarge` returns true if the node has too many elements and must be split.

# Insertion

It's time to implement insertion. We'll follow the same procedure we outlined in the introduction of this chapter; we just need to translate it into working Swift code.

We'll start by defining a simple struct that consists of an element and a node. I like to call this construct a *splinter*:

```
extension BTree {
  struct Splinter {
    let separator: Element
    let node: Node
  }
}
```

The `separator` value of a splinter must be less than any of the elements inside its `node`. A splinter is like a thin slice of a node: it consists of a single element and the child node that immediately follows it. Hence the name!

Next, we'll define a method that splits a node in half, extracting half of the elements into a new splinter:

```
extension BTree.Node {
```

```
  func split() -> BTree<Element>.Splinter {
    let count = self.elements.count
    let middle = count / 2

    let separator = self.elements[middle]

    let node = BTree<Element>.Node(order: order)
    node.elements.append(contentsOf: self.elements[middle + 1 ..< count])
    self.elements.removeSubrange(middle ..< count)

    if !isLeaf {
      node.children.append(contentsOf: self.children[middle + 1 ..< count + 1])
      self.children.removeSubrange(middle + 1 ..< count + 1)
    }
    return .init(separator: separator, node: node)
  }
}
```

The only tricky aspect in this function is index management: it's easy to make an off-by-one error somewhere and leave the tree broken.

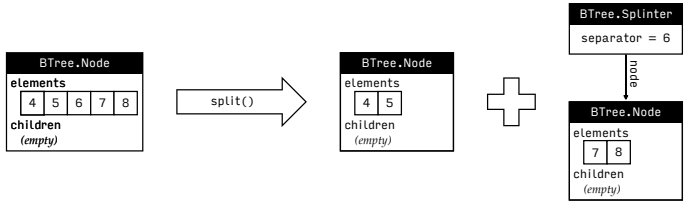



**Figure 6.2: The result of splitting an oversized node with values 4--8. The split() operation returns a splinter with a new node containing elements 7 and 8, leaving the original node with elements 4 and 5.**

Let's see how we can use splinters and splitting to insert a new element into a node:

```
extension BTree.Node {
  func insert(_ element: Element) -> (old: Element?, splinter: BTree<Element>.Splinter?) {
```

The method starts by finding the slot in the node corresponding to the new element. If the element is already there, it just returns the value without modifying anything:

```
    let slot = self.slot(of: element)
    if slot.match {
      // The element is already in the tree.
      return (self.elements[slot.index], nil)
    }
```

Otherwise, it needs to actually insert it, so it must increment the mutation counter:

```
    mutationCount += 1
```

Inserting a new element into a leaf node is simple: we just need to insert it at the correct slot in the elements array. However, the extra element may make the node too big. When that happens, we cut it in half using split() and return the resulting splinter:

```
    if self.isLeaf {
      elements.insert(element, at: slot.index)
      return (nil, self.isTooLarge ? self.split() : nil)
    }
```

If the node has children, then we need to make a recursive call to insert it into the child at the correct slot. insert is a mutating method, so we must call makeChildUnique(at:) to make a COW copy when necessary. If the recursive insert call returns a splinter, we must insert it into self at the slot we already calculated:

```
    let (old, splinter) = makeChildUnique(at: slot.index).insert(element)
    guard let s = splinter else { return (old, nil) }
    elements.insert(s.separator, at: slot.index)
    children.insert(s.node, at: slot.index + 1)
    return (nil, self.isTooLarge ? self.split() : nil)
  }
}
```

So if we insert an element with a path in the B-tree that has full nodes at its end, then insert triggers a cascade of splits, propagating upward from the insertion point toward the root of the tree.

If all nodes on the path are full, then the root node itself gets split. When that happens, we need to add an extra level to the tree — which is done by creating a new root node one level above the previous one — containing the old root and the splinter. The best place to do this is in the public insert method inside the BTree struct, though we must remember to call makeRootUnique before mutating anything in the tree:

```
extension BTree {
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪ Element) {
    let root = makeRootUnique()
    let (old, splinter) = root.insert(element)
    if let splinter = splinter {
      let r = Node(order: root.order)
      r.elements = [splinter.separator]
      r.children = [root, splinter.node]
      self.root = r
    }
    return (old == nil, old ?? element)
  }
}
```

That's all it takes to insert elements into a B-tree. The code is definitely not as elegant as RedBlackTree, but I think it's a great deal nicer than what we had in RedblackTree2.

Benchmarking insertion performance gets us the chart in figure 6.3. For small sets, we're now definitely within shooting distance of SortedArray: BTree.insert is just 10–15% slower. Even better, for large sets, the same code also improves on RedblackTree2's insertion speed by a factor of 3.5! Creating a BTree by randomly inserting 4 million individual elements takes just three seconds.

As a balanced search tree, B-tree.insert has the same $O(\log n)$ asymptotic performance as RedBlackTree and RedblackTree2, but it also incorporates the memory access patterns of SortedArray, leading to a dramatic speedup for all input sizes, big and small.

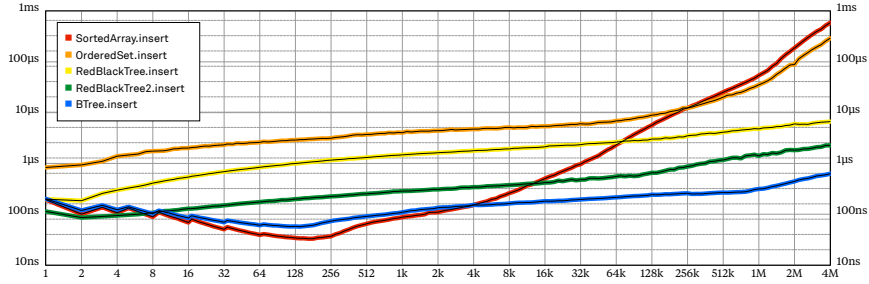



**Figure 6.3: Comparing the performance of five different implementations for SortedSet.insert.**

Sometimes combining two data structures gets us a third that's more than just the sum of its parts!

# Implementing Collection

The biggest design challenge in implementing `Collection` is to come up with a nice index type. For B-trees, we're going to follow in `RedblackTree2`'s footsteps and have each index contain a full path inside the tree.

## B-Tree Paths

A path inside a B-tree can be represented by a series of slots starting with the root node. But to make things more efficient, we'll also include references to all the nodes along the path.

Indices must not contain strong references to parts of the collection. In `RedblackTree2`, we initially satisfied this requirement using zeroing weak references. This time, we'll apply what we learned at the end of the last chapter, and we'll use unsafe unowned references instead. As we've seen, these have no reference counting overhead, so they're a tiny bit faster. This small advantage really adds up in our iteration microbenchmark: for `RedblackTree2`, it led to a 200% speedup.

In exchange for raw performance, the language provides absolutely no safety net for dereferencing `unowned(unsafe)` references; their targets may not exist anymore, or they

may contain some data that's not what we expect. They work similar to C pointers in this respect.

This isn't a problem for valid indices, as their trees are still in the exact same state as they were when they were created. But we have to be extra careful with invalid indices, as they may have broken node references on their path. Looking at the target of a broken record results in undefined behavior. The app may crash, or it may silently produce corrupt data. (This is different than simple `unowned` references, which still do a little bit of reference counting to ensure a guaranteed trap when their target has disappeared.)

Let's start writing code. We'll represent a B-tree path using an array of `UnsafePathElements`, which we define as a struct holding a node reference and a slot number:

```
extension BTree {
  struct UnsafePathElement {
    unowned(unsafe) let node: Node
    var slot: Int

    init(_ node: Node, _ slot: Int) {
      self.node = node
      self.slot = slot
    }
  }
}
```

That `node` there is one scary-looking declaration for a stored property.

We'll also define a handful of computed properties for accessing common aspects of a path element, including the value it references, the corresponding child node preceding this value, whether the associated node is a leaf, and whether the path element is pointing at the end of the node:

```
extension BTree.UnsafePathElement {
  var value: Element? {
    guard slot < node.elements.count else { return nil }
    return node.elements[slot]
  }
```

```
  var child: BTree<Element>.Node {
    return node.children[slot]
  }
  var isLeaf: Bool { return node.isLeaf }
  var isAtEnd: Bool { return slot == node.elements.count }
}
```

Note that the slot may be the one immediately after the last element inside the node, so a path element may not always have a corresponding value. Internal nodes have a child node at every slot though.

It'll be useful to be able to compare path elements for equality, so let's also implement Equatable:

```
extension BTree.UnsafePathElement: Equatable {
  static func ==(left: BTree<Element>.UnsafePathElement, right:
  ↪ BTree<Element>.UnsafePathElement) -> Bool {
    return left.node === right.node && left.slot == right.slot
  }
}
```

## B-Tree Indices

Here's the definition of BTreeIndex. It looks similar to RedblackTree2Index, in that it just holds a list of path elements, along with a weak reference to the root node and mutation count at the time the index was created. Note that we still use a weak reference to the root node — because it allows us to safely validate an index on its own — when we have no concrete tree value to match with the index:

```
extension BTree {
  public struct Index {
    fileprivate weak var root: Node?
    fileprivate let mutationCount: Int64

    fileprivate var path: [UnsafePathElement]
    fileprivate var current: UnsafePathElement
```

In most cases, advancing an index inside a B-tree can be done by simply incrementing the slot value of the final path element. Storing this "hot" element outside the array as a separate `current` property makes this common case a tiny bit faster. (Array's own index validation and the inherent indirection in accessing the array's underlying storage buffer would add a little bit of extra overhead.) Such micro-optimizations are normally unnecessary (or even harmful). However, we've already decided to use unsafe references, and this additional complication seems almost trivial in comparison; it's certainly less dangerous.

We also need two internal initializers for creating the `startIndex` and `endIndex` of a tree. The former constructs the path to the first element inside the tree, while the latter just puts the path after the last element in the root node:

```
    init(startOf tree: BTree) {
      self.root = tree.root
      self.mutationCount = tree.root.mutationCount
      self.path = []
      self.current = UnsafePathElement(tree.root, 0)
      while !current.isLeaf { push(0) }
    }

    init(endOf tree: BTree) {
      self.root = tree.root
      self.mutationCount = tree.root.mutationCount
      self.path = []
      self.current = UnsafePathElement(tree.root, tree.root.elements.count)
    }
  }
}
```

In theory, `startIndex` is an $O(\log n)$ operation; but as we've seen, the depth of a B-tree is more like $O(1)$ in practice, so in this case, we aren't really breaking `Collection`'s performance requirement at all.

## Index Validation

Index validation is done in basically the same way as in `RedblackTree2`, except the root reference may not be `nil` in any valid B-tree index. Even the empty tree has a root node.

All mutations on BTree values are careful to change either the root reference or the root's mutation count so that when both of these match the values in an index, we know the index is still valid and it's safe to look at its path:

```
extension BTree.Index {
  fileprivate func validate(for root: BTree<Element>.Node) {
    precondition(self.root === root)
    precondition(self.mutationCount == root.mutationCount)
  }

  fileprivate static func validate(_ left: BTree<Element>.Index, _ right:
  ↪ BTree<Element>.Index) {
    precondition(left.root === right.root)
    precondition(left.mutationCount == right.mutationCount)
    precondition(left.root != nil)
    precondition(left.mutationCount == left.root!.mutationCount)
  }
}
```

We'll need the static index validation method in our Equatable and Comparable implementations. This method is the reason why we couldn't change the weak reference to the root node into unowned(unsafe) — it needs to be able to validate indices without an externally supplied tree context.

## Index Navigation

To navigate inside the tree, we define two helper methods, push and pop. push takes a slot number in the child node associated with the current path and appends it to the end of the path. pop simply removes the last element on the path:

```
extension BTree.Index {
  fileprivate mutating func push(_ slot: Int) {
    path.append(current)
    let child = current.node.children[current.slot]
    current = BTree<Element>.UnsafePathElement(child, slot)
  }
```

```
  fileprivate mutating func pop() {
    current = self.path.removeLast()
  }
}
```

Given these two functions, we can define a method that advances an index to the next element inside the tree. For the vast majority of cases, the method just needs to increment the slot of the current (i.e. last) path element. The only exceptions are when (1) there are no more elements in the corresponding leaf node, or (2) the current node isn't a leaf. (Both of these cases are relatively rare.)

```
extension BTree.Index {
  fileprivate mutating func formSuccessor() {
    precondition(!current.isAtEnd, "Cannot advance beyond endIndex")
    current.slot += 1
    if current.isLeaf {
      // This loop will rarely execute even once.
      while current.isAtEnd, current.node !== root {
        // Ascend to the nearest ancestor that has further elements.
        pop()
      }
    }
    else {
      // Descend to the start of the leftmost leaf node under us.
      while !current.isLeaf {
        push(0)
      }
    }
  }
}
```

Finding the predecessor index is similar, although we need to slightly reorganize the code because we need to look out for the start of the node and not the end:

```
extension BTree.Index {
  fileprivate mutating func formPredecessor() {
    if current.isLeaf {
```

```
        while current.slot == 0, current.node !== root {
          pop()
        }
        precondition(current.slot > 0, "Cannot go below startIndex")
        current.slot -= 1
      }
      else {
        while !current.isLeaf {
          let c = current.child
          push(c.isLeaf ? c.elements.count - 1 : c.elements.count)
        }
      }
    }
}
```

The functions above are all private helper methods, so it's OK for them to assume that the index has already been validated by their caller.

## Comparing Indices

Indices must be comparable, so we need to implement the equality and less-than operators too. Since these functions are public entry points, we must remember to validate indices before accessing any of their path elements:

```
extension BTree.Index: Comparable {
  public static func ==(left: BTree<Element>.Index, right: BTree<Element>.Index) -> Bool {
    BTree<Element>.Index.validate(left, right)
    return left.current == right.current
  }

  public static func <(left: BTree<Element>.Index, right: BTree<Element>.Index) -> Bool {
    BTree<Element>.Index.validate(left, right)
    switch (left.current.value, right.current.value) {
    case let (a?, b?): return a < b
    case (nil, _): return false
    default: return true
    }
```

```
  }
}
```

## Collection Conformance

We now have everything we need to make BTree conform to BidirectionalCollection; in the implementation of each method, we just need to call one of the index methods above. We also need to remember to validate any indices before calling anything that doesn't implement validation on its own:

```
extension BTree: SortedSet {
  public var startIndex: Index { return Index(startOf: self) }
  public var endIndex: Index { return Index(endOf: self) }

  public subscript(index: Index) -> Element {
    index.validate(for: root)
    return index.current.value!
  }

  public func formIndex(after i: inout Index) {
    i.validate(for: root)
    i.formSuccessor()
  }

  public func formIndex(before i: inout Index) {
    i.validate(for: root)
    i.formPredecessor()
  }

  public func index(after i: Index) -> Index {
    i.validate(for: root)
    var i = i
    i.formSuccessor()
    return i
  }

  public func index(before i: Index) -> Index {
    i.validate(for: root)
```

```
    var i = i
    i.formPredecessor()
    return i
  }
}
```

## Counting the Number of Elements

Iteration using indexing only takes $O(n)$ time now, but all that index validation may slow things down quite a bit. As such, it makes sense to include specialized implementations for basic `Collection` members, like the `count` property:

```
extension BTree {
  public var count: Int {
    return root.count
  }
}

extension BTree.Node {
  var count: Int {
    return children.reduce(elements.count) { $0 + $1.count }
  }
}
```

Note that `Node.count` uses recursion inside the closure it gives to `reduce`. It needs to visit every node inside the B-tree, and because we technically have $O(n)$ nodes, the count will finish in $O(n)$ time. (This is still much faster than counting each element one by one though.)

> Since B-tree nodes are so big, it would make sense to include a subtree count inside each node, making the data structure an *order statistic tree*. This would make `count` an $O(1)$ operation, and it would allow looking up the $i$th element inside the tree in just $O(\log n)$ time. We won't implement this here; check out the official BTree package for a full implementation of this idea.

## Custom Iterator

We should also define a custom iterator type in order to eliminate index validation overhead in simple for-in loops. The following is a straightforward adaptation of the one we made for RedblackTree2:

```
extension BTree {
  public struct Iterator: IteratorProtocol {
    let tree: BTree
    var index: Index

    init(_ tree: BTree) {
      self.tree = tree
      self.index = tree.startIndex
    }

    public mutating func next() -> Element? {
      guard let result = index.current.value else { return nil }
      index.formSuccessor()
      return result
    }
  }

  public func makeIterator() -> Iterator {
    return Iterator(self)
  }
}
```

## Iteration Performance

We used every tool in our toolbox to optimize the performance of iteration inside B-trees:

- → **Algorithmic improvement:** Indices are represented by full paths into the tree so that index advancement can be implemented in amortized $O(1)$ time.
- → **Optimizing for the typical case:** The vast majority of values inside B-trees are stored in the `elements` arrays of leaf nodes. By extracting the last path element

into its own stored property, we ensured we advance over such elements using as few operations as possible.

→ **Improved locality of reference:** By arranging our values into arrays with a size that's a function of the cache size, we built a data structure that's perfectly adapted to the multi-layered memory architecture in our computers.

→ **Validation shortcuts:** We implemented a custom iterator so that we can eliminate redundant index validation during iteration.

→ **Careful use of unsafe operations:** Inside paths, we use unsafe references to tree nodes so that creating/modifying index paths doesn't involve reference count operations. Index validation ensures we never encounter broken references.

So how fast is iteration over B-trees? Pretty darned fast! Figure 6.4 has our microbenchmark results.

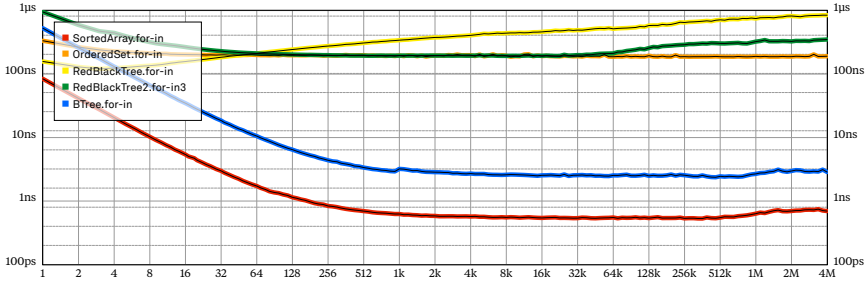



**Figure 6.4: Comparing the performance of iteration in five SortedSet implementations.**

We were quite happy with RedblackTree2's final iteration performance; however, BTree is in a completely different league: it's about 40 times faster! SortedArray is 8 times faster still though, so there's some room for improvement.

# Examples

```
var set = BTree<Int>(order: 5)
for i in (1 ... 250).shuffled() {
  set.insert(i)
```

```
}
▶ set
  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
↪  23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
↪  42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
↪  61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
↪  80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98,
↪  99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113,
↪  114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
↪  129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
↪  144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158,
↪  159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173,
↪  174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188,
↪  189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203,
↪  204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218,
↪  219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233,
↪  234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248,
↪  249, 250]


let evenMembers = set.reversed().lazy.filter { $0 % 2 == 0 }.map { "\($0)"
↪  }.joined(separator: ", ")
▶ evenMembers
  250, 248, 246, 244, 242, 240, 238, 236, 234, 232, 230, 228, 226, 224, 222,
↪  220, 218, 216, 214, 212, 210, 208, 206, 204, 202, 200, 198, 196, 194, 192,
↪  190, 188, 186, 184, 182, 180, 178, 176, 174, 172, 170, 168, 166, 164, 162,
↪  160, 158, 156, 154, 152, 150, 148, 146, 144, 142, 140, 138, 136, 134, 132,
↪  130, 128, 126, 124, 122, 120, 118, 116, 114, 112, 110, 108, 106, 104, 102,
↪  100, 98, 96, 94, 92, 90, 88, 86, 84, 82, 80, 78, 76, 74, 72, 70, 68, 66, 64,
↪  62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26,
↪  24, 22, 20, 18, 16, 14, 12, 10, 8, 6, 4, 2
```

# Performance Summary

Figure 6.5 summarizes the performance of BTree on our four standard microbenchmarks. Remarkably, iteration using for-in is now much faster than using forEach — this is the opposite of what we had with red-black trees.

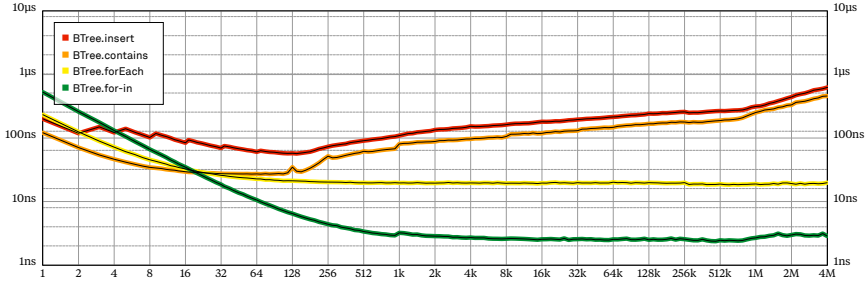



**Figure 6.5: Benchmark results for BTree operations.**

Our benchmarks use `Int` elements on a 64-bit MacBook with an L1 cache of 32 kilobytes — so `BTree<Int>` has a default order of 1,024. Therefore, the first node split happens when we insert the 1,024th element, converting the tree from a single node into three nodes arranged in two levels. This point is clearly visible on the chart as a sudden jump at the 1,000 size.

The size at which the tree grows a third level isn't as precisely predetermined — it happens somewhere between 500,000 and 1 million items. The curves for `insert` and `contains` exhibit faster growth at this range, which is consistent with having to recurse a level deeper.

To get four-level B-trees, we need to insert about 1 billion integers. Figure 6.6 extends the `BTree.insert` chart to such monstrously huge sets. The insertion curve has three clearly distinguishable phases corresponding to B-trees of one, two, and three levels, and at the right edge of the chart we can clearly make out the beginnings of the upward curve that marks the fourth tree layer. At this point though, the benchmark has eaten up all 16 GB of physical memory in my MacBook, forcing macOS to start compressing pages and even paging out some of the data to the SSD. Clearly, some (most?) of that final growth spurt is due to paging and not the addition of the fourth B-tree layer. Extending the benchmark even further to the right would push beyond the limits of this particular machine.

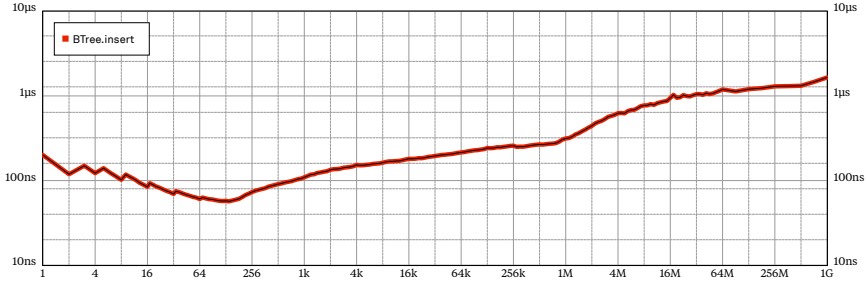



**Figure 6.6: Creating a B-tree by random insertions.**

# Additional Optimizations

7

In this chapter, we're going to concentrate on optimizing BTree.insert even further, squeezing our code to get the last few drops of sweet performance out.

We'll create three more SortedSet implementations, creatively naming them BTree2, BTree3, and BTree4. To keep this book at a manageable length, we aren't going to include the full source code of these three B-tree variants; we'll just describe the changes through a few representative code snippets. You can find the full source code for all three B-tree variants in the book's GitHub repository.

If you're bored with SortedSets, it's OK to skip this chapter, as the advanced techniques it describes are rarely applicable to everyday app development work.

# Inlining Array's Methods

BTree stores elements and children of a Node in standard Arrays. In the last chapter, this made the code relatively easy to understand, which helped us explain B-trees. However, Array includes logic for index validation and COW that is redundant inside BTree — supposing we didn't make any coding mistakes, BTree never subscripts an array with an out-of-range index, and it implements its own COW behavior.

Looking at our insertion benchmark chart (figure 7.1), we see that while BTree.insert is extremely close to SortedArray.insert at small set sizes, there's still a 10–20% performance gap between them. Would eliminating Array's (tiny) overhead be enough to close this gap? Let's find out!

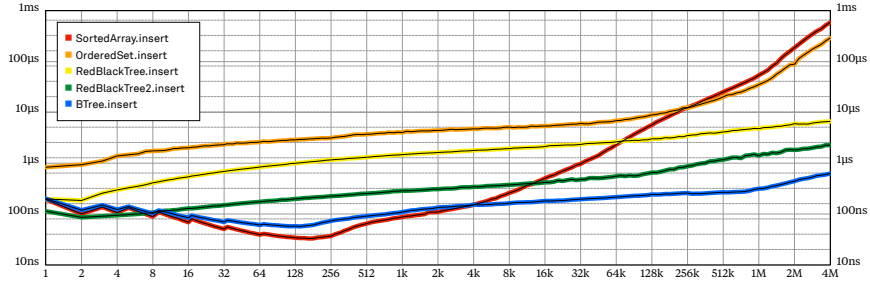



**Figure 7.1: Comparing the performance of five different implementations for SortedSet.insert.**

The Swift standard library includes UnsafeMutablePointer and UnsafeMutableBufferPointer types we can use to implement our own storage buffers. They deserve their scary names; working with these types is only slightly easier than working with C pointers — the slightest mistake in the code that handles them may result in hard-to-debug memory corruption bugs, memory leaks, or even worse! On the other hand, if we promise to be super careful, we might be able to use these to slightly improve performance.

So let's start writing BTree2, our second attempt at implementing B-trees. It starts innocently enough:

```
public struct BTree2<Element: Comparable> {
  fileprivate var root: Node

  public init(order: Int) {
    self.root = Node(order: order)
  }
}
```

However, in Node, we convert the elements array into an unsafe mutable pointer that points to the start of a manually allocated buffer. We also need to add a stored property for the count of elements currently in the buffer; the pointer won't keep track of this on its own:

```
extension BTree2 {
  class Node {
    let order: Int
    var mutationCount: Int64 = 0
    var elementCount: Int = 0
    let elements: UnsafeMutablePointer<Element>
    var children: ContiguousArray<Node> = []
```

For BTree2, we left children as an array, although we did change its type from Array to ContiguousArray; the latter may sometimes be slightly faster. Remember that most elements are in leaf nodes — speeding up internal nodes may not result in a measurable overall improvement, so it's better not to waste too much time on optimizing them.

Node's designated initializer is responsible for allocating our element buffer. We allocate space for order elements so that the buffer will be able to hold one more element than our maximum size. This is important, because nodes will grow over the maximum just before we split them:

```
    init(order: Int) {
      self.order = order
      self.elements = .allocate(capacity: order)
    }
```

In the previous chapter, we didn't use Array.reserveCapacity(_:) to preallocate storage for the maximum sizes of our two arrays, relying instead on Array's automatic storage management. This makes the code simpler, but it has two consequences. First, when we insert a new element into a BTree node, Array sometimes needs to allocate a new, larger buffer and move existing elements into it. This adds some additional overhead to insert operations. Second, Array grows its storage buffer in powers of two, so if the tree order is just *slightly* above such a number, Array may allocate as much as 50% more space than we need for a full node. We eliminate both of these issues here by always allocating a buffer that's exactly as large as our maximum node size.

Since we're manually allocating the buffer, we need to manually deallocate it at some point. The best place to do this is in a custom deinit:

```
    deinit {
      elements.deinitialize(count: elementCount)
      elements.deallocate(capacity: order)
    }
  }
}
```

Note how we have to explicitly deinitialize memory that's occupied by elements, returning the buffer to its initial, uninitialized state before deallocating it. This ensures that references that may be included in Element values are correctly released, possibly even deallocating some values that were only held by the buffer. Neglecting to deinitialize values will result in memory leaks:

Cloning a node is a nice example of how we may use the elements buffer. We call the initialize(from:count:) method to copy data between buffers. This takes care of updating

reference counts if necessary. If the node is an internal node, we also call reserveCapacity(_:) on the new node's children array to preallocate enough space for holding as many children as we may need. Otherwise, we leave children empty so that we don't waste memory allocating storage we'll never use:

```
extension BTree2.Node {
  func clone() -> BTree2<Element>.Node {
    let node = BTree2<Element>.Node(order: order)
    node.elementCount = self.elementCount
    node.elements.initialize(from: self.elements, count: self.elementCount)
    if !isLeaf {
      node.children.reserveCapacity(order + 1)
      node.children += self.children
    }
    return node
  }
}
```

The split() operation we need for insertion can take advantage of UnsafeMutablePointer's support for move initialization. In the original code, we had to move elements in two phases by first copying them into the new array and then removing them from the original array. The unified move operation can be much faster when Element includes reference-counted values. (It doesn't affect performance for simple value types like Int though.)

```
extension BTree2.Node {
  func split() -> BTree2<Element>.Splinter {
    let count = self.elementCount
    let middle = count / 2

    let separator = elements[middle]
    let node = BTree2<Element>.Node(order: self.order)

    let c = count - middle - 1
    node.elements.moveInitialize(from: self.elements + middle + 1, count: c)
    node.elementCount = c
    self.elementCount = middle
```

```
      if !isLeaf {
        node.children.reserveCapacity(self.order + 1)
        node.children += self.children[middle + 1 ... count]
        self.children.removeSubrange(middle + 1 ... count)
      }
      return .init(separator: separator, node: node)
    }
}
```

To insert a new element in the middle of our buffer, we have to implement the equivalent of Array.insert. To do this, we first have to make room for the new element by moving existing items one slot to the right, starting at the insertion point:

```
extension BTree2.Node {
  fileprivate func _insertElement(_ element: Element, at slot: Int) {
    assert(slot >= 0 && slot <= elementCount)
    (elements + slot + 1).moveInitialize(from: elements + slot, count: elementCount -
    ↪ slot)
    (elements + slot).initialize(to: element)
    elementCount += 1
  }
}
```

This is all we need in order to adapt our old insert code for BTree2:

```
extension BTree2.Node {
  func insert(_ element: Element) -> (old: Element?, splinter: BTree2<Element>.Splinter?)
  ↪ {
    let slot = self.slot(of: element)
    if slot.match {
      // The element is already in the tree.
      return (self.elements[slot.index], nil)
    }
    mutationCount += 1
    if self.isLeaf {
      _insertElement(element, at: slot.index)
      return (nil, self.isTooLarge ? self.split() : nil)
```

```
      }
      let (old, splinter) = makeChildUnique(at: slot.index).insert(element)
      guard let s = splinter else { return (old, nil) }
      _insertElement(s.separator, at: slot.index)
      self.children.insert(s.node, at: slot.index + 1)
      return (old, self.isTooLarge ? self.split() : nil)
    }
}

extension BTree2 {
  @discardableResult
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪ Element) {
    let root = makeRootUnique()
    let (old, splinter) = root.insert(element)
    if let s = splinter {
      let root = BTree2<Element>.Node(order: root.order)
      root.elementCount = 1
      root.elements.initialize(to: s.separator)
      root.children = [self.root, s.node]
      self.root = root
    }
    return (inserted: old == nil, memberAfterInsert: old ?? element)
  }
}
```

To create BTree2, we essentially manually inlined Array's implementation from the standard library directly into our B-tree code, removing functionality that we don't need.

Benchmarking results are shown in figure 7.2. Insertion has consistently become 10–20% faster. We've eliminated the last remaining performance gap between B-trees and sorted arrays: BTree2.insert matches or exceeds the performance of all previous SortedSet implementations across the entire size spectrum.

As a nice side effect, eliminating Array's index validation checks also boosts iteration performance by a factor of two; see figure 7.3. BTree2.for-in is just four times slower than SortedArray; that's a pretty sizable improvement!

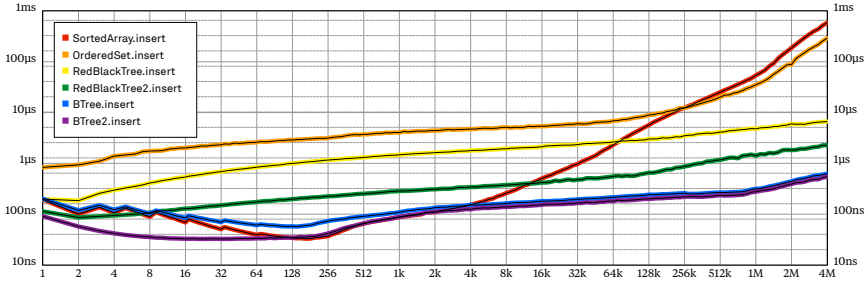



**Figure 7.2: Comparing insertion performance of six SortedSet implementations.**

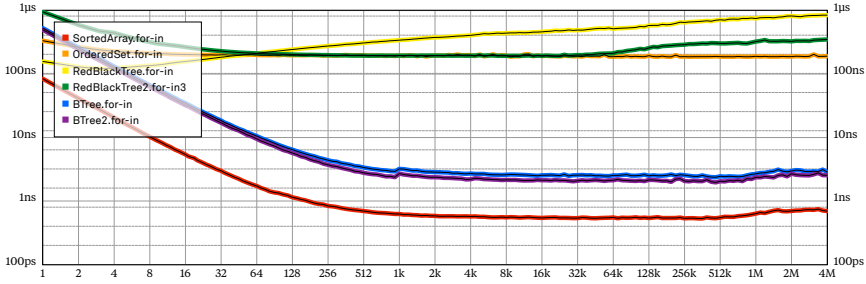



**Figure 7.3: Comparing the iteration performance of six SortedSet implementations.**

# Optimizing Insertion into Shared Storage

So far, whenever we've measured insertion performance, we've always assumed the storage of our sorted set is entirely unique to it. But we never benchmarked what happens when we insert elements into sets with shared storage.

Let's devise a benchmark for measuring shared insertions! One way to do this is to measure how much time it takes to insert a bunch of elements when we make a full copy of the entire set after every insertion:

```
extension SortedSet {
  func sharedInsertBenchmark(_ input: [Element]) {
    var set = Self()
    var copy = set
    for value in input {
      set.insert(value)
      copy = set
    }
    _ = copy // Prevents warning about the variable never being read.
  }
}
```

Figure 7.4 displays the results of running this new benchmark on the SortedSet implementations we've made thus far.

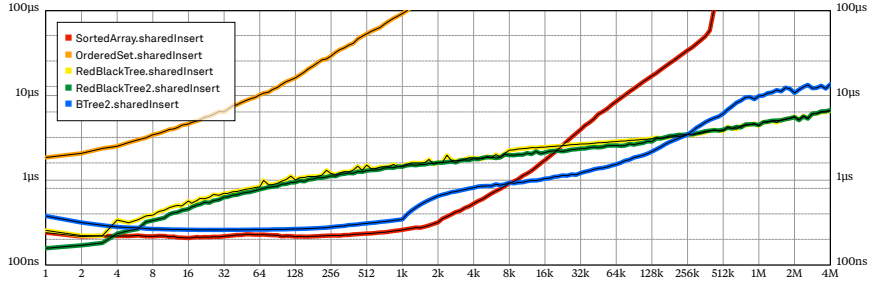



**Figure 7.4: Amortized time of a single insertion into shared storage.**

Clearly, our NSOrderedSet wrapper wasn't built for this kind of abuse; once we pass a couple thousand values, it becomes roughly a thousand times slower than SortedArray. And SortedArray doesn't fare much better either: to dissociate their storage from their copies, both of these array-based sorted sets need to make a full copy of every single value stored in them. This doesn't change their insertion implementation's asymptotic performance (it's still $O(n)$), but it does add a sizeable constant factor. In the case of SortedArray, the sharedInsert benchmark is about 3.5 times slower than plain insert.

Our two red-black tree implementations behave much better: for each insertion, they only need to make a copy of nodes that happen to fall on the path toward the newly inserted element. RedBlackTree does this anyway: its insertion performance is exactly the same whether or not its nodes are shared. But RedblackTree2 is unable to use its fancy in-place mutations: all of its isKnownUniquelyReferenced calls return false, so it becomes slightly slower than RedBlackTree at this particular benchmark.

BTree2 starts off really promising, but it slows down rather dramatically at around 64,000 elements. At this stage, the tree is getting close to growing a third level; its (second-level) root node contains enough elements that having to make a copy of it considerably weighs down insertions. This gets much worse as the tree gains its third level, ending up on a performance level that's about 6 times slower than red-black trees. (BTree.insert's asymptotic performance remains $O(\log n)$; the slowdown just adds a huge constant factor.)

We want our B-trees to be much faster than red-black trees on every chart. Can we somehow prevent this bump from happening in the shared storage case? Well, of course we can!

Our theory is that the slowdown occurs because of large internal nodes. We also know that internal nodes *usually* don't contribute much to B-tree performance, because most values are stored in leaf nodes. Therefore, we're probably able to mess with internal nodes whichever way we like — it won't affect the charts much. So what if we drastically restricted the maximum size of our internal nodes while leaving leaf nodes alone?

It turns out this is shockingly easy to do: we just need to make a tiny one-line change in BTree2.insert:

```
extension BTree3 {
  @discardableResult
```

```
  public mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪   Element) {
    let root = makeRootUnique()
    let (old, splinter) = root.insert(element)
    if let s = splinter {
      let root = BTree3<Element>.Node(order: 16) // <--
      root.elementCount = 1
      root.elements.initialize(to: s.separator)
      root.children = [self.root, s.node]
      self.root = root
    }
    return (inserted: old == nil, memberAfterInsert: old ?? element)
  }
}
```

The marked line is inside the code block that adds a new level to the tree. The order number we use for the new root also gets applied to all nodes that'll be split off of it — so by using a small order number here instead of `self.order`, we ensure that *all* internal nodes get initialized with this order instead of with the value originally given to `BTree`'s initializer. (New leaf nodes are always split off from existing leaves, so this number won't ever get applied to them.)

Naming this new variant `BTree3` and rerunning our benchmark gets us the chart in figure 7.5. Our theory was correct: limiting the size of internal nodes did in fact eliminate the slowdown! `BTree3` is about 2–2.5 times faster than `RedBlackTree`, even for huge sets. (Creating a `BTree3` of 4 million elements in this laborious manner takes just 15 seconds; `BTree2` took 10 times as long.)

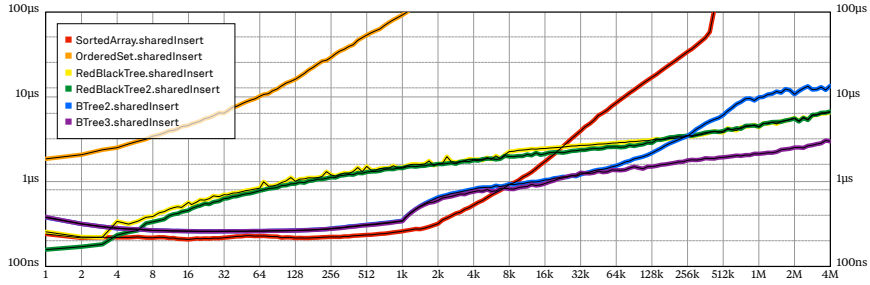



**Figure 7.5: Amortized time of a single insertion into shared storage.**

Limiting the size of internal nodes increases the typical height of the tree, so it does affect some B-tree operations. Thankfully though, the effect is mostly negligible: it slows down `contains` and the in-place mutating flavor of `insert` only by about 10%, and it doesn't affect iteration methods at all. For instance, figure 7.6 compares `BTree3`'s in-place insertion performance to our other implementations.

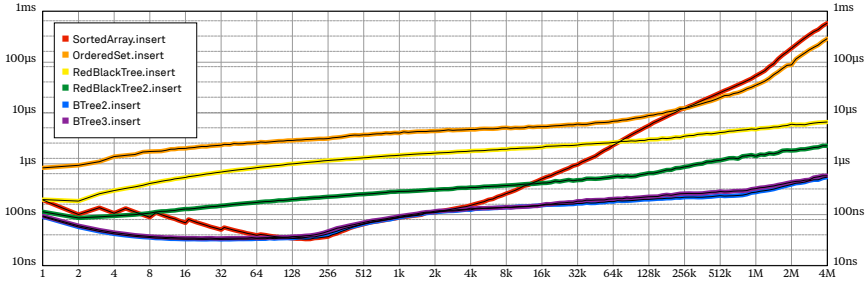



**Figure 7.6: Amortized time of a single insertion into uniquely held storage.**

That was surprisingly easy, wasn't it?

# Eliminating Redundant Copying

Whenever we had to implement COW, we did it by implementing `makeFooUnique` methods that create clones of shared storage when necessary. Our code then proceeded to mutate the resulting unique storage in a separate, second step.

If you think about it, this is a rather inefficient way of doing this: for example, when we have to insert a new element at the start of a shared `OrderedSet` value, `makeUnique` first makes a brand-new `NSOrderedSet` with the exact same elements, and then `insert` proceeds to immediately move all of them one slot to the right to make room for the new item.

No wonder `sharedInsert` was so slow for `OrderedSets`! It would be much faster to create the clone of the shared storage such that it already has the new element inserted at the correct place. We won't implement this for `OrderedSet` though, as it isn't worth the effort: it would still remain far behind our other `SortedSet` implementations.

However, we can observe the same suboptimal behavior in all three `BTree` variants we've made thus far. When we insert an element into a shared B-tree node, we start by copying all existing elements into a new buffer; we insert the new value in a separate step. What's even worse is the insertion may result in an oversized node, which then needs to be split — and this is done as yet another separate operation, moving half of the elements into a third buffer. A single insertion may therefore copy/move several hundred elements not once, not twice, but *three different times*! That seems inefficient.

It's possible to eliminate all this needless copying/moving by merging `makeUnique`, `insert`, and `split` into a single, rather complicated operation. Merging `makeUnique` and `insert` can be done by implementing a non-mutating variant of insertion and switching to it when we detect the node is shared. To unify this hybrid insertion with `split`, we need to detect that a split will be necessary *before* doing the insertion, and if so, create two nodes from scratch so that the new element is already in the correct place. Note that the new element may end up inside the first node, or inside the second node, or — if it happens to fall exactly in the middle — it may become the separator value between the two halves of the original node. Thus, to unify all three operations, we need to write $2 \times 4 = 8$ separate variants of insertion.

I did my best to implement this; unfortunately, the result was a wash. Shared insertion became a little bit faster for certain element counts, and it became a little bit slower for others. In-place insertion didn't change much at all.

I won't include the code for my attempt for `BTree4` here, but you'll find it in the GitHub repository. Try tweaking it; maybe you'll succeed at making it meaningfully faster!

# Conclusion

8

In this book, we've discussed seven different ways to implement the same simple collection type. Every time we created a new solution, our code became incrementally more complicated, but in exchange, we gained substantial performance improvements. This is perhaps best demonstrated by the steady downward progress of our successive insertion implementations on the benchmark chart.

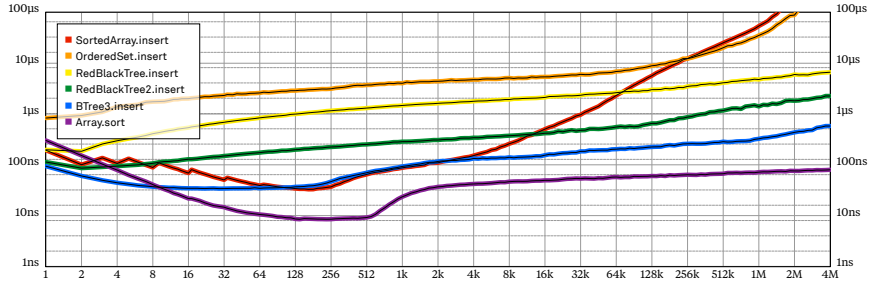



**Figure 8.1: Comparing insertion performance of five SortedSet implementations to the amortized per-element cost of Array.sort.**

Is it possible to implement an even faster SortedSet.insert? Well, BTree3 certainly has room for some additional micro-optimizations; I think a 5–10% improvement is definitely possible. Investing even more effort would perhaps get us 20%.

But is it possible we missed some clever optimization trick that would get us another huge performance jump of 200–400%? I don't believe so.

First of all, note that by inserting a bunch of elements into a sorted set, we're essentially sorting them. We could also do that by simply calling Array.sort, which implements the super speedy Introsort sorting algorithm. The last line in the chart above depicts the amortized time Array.sort spends on each element. In a very real sense, Array.sort sets a hard upper limit on the performance we can expect out of any sorted set.

At typical sizes, sorting elements by calling BTree3.insert in a loop is *just 3.5 times slower* than Array.sort. This is already *amazingly* close! Consider that the BTree3 benchmark processes each element individually and keeps existing elements neatly sorted after each insertion. I find it surprising that BTree3.insert gets so close despite such a huge handicap, and I'd be shocked to learn about a new SortedSet implementation that improves on B-trees by even 50%.

# Implementing Insertion in Constant Time

Although it may not be possible to drastically improve on BTree3 while fully implementing all SortedSet requirements, we can always speed things up by cheating!

For example, SillySet in the code below implements the syntactic requirements of the SortedSet protocol with an insert method that takes $O(1)$ time. It runs circles around Array.sort in the insert benchmark above without even breaking a sweat:

```
struct SillySet<Element: Hashable & Comparable>: SortedSet, RandomAccessCollection {
  typealias Indices = CountableRange<Int>

  class Storage {
    var v: [Element]
    var s: Set<Element>
    var extras: Set<Element> = []

    init(_ v: [Element]) {
      self.v = v
      self.s = Set(v)
    }

    func commit() {
      guard !extras.isEmpty else { return }
      s.formUnion(extras)
      v += extras
      v.sort()
      extras = []
    }
  }

  private var storage = Storage([])

  var startIndex: Int { return 0 }

  var endIndex: Int { return storage.s.count + storage.extras.count }
```

```
  // Complexity: `O(n*log(n))`, where `n` is the number of insertions since the last time
  ↪ `subscript` was called.
  subscript(i: Int) -> Element {
    storage.commit()
    return storage.v[i]
  }

  // Complexity: O(1)
  func contains(_ element: Element) -> Bool {
    return storage.s.contains(element) || storage.extras.contains(element)
  }

  // Complexity: O(1) unless storage is shared.
  mutating func insert(_ element: Element) -> (inserted: Bool, memberAfterInsert:
  ↪ Element) {
    if !isKnownUniquelyReferenced(&storage) {
      storage = Storage(storage.v)
    }
    if let i = storage.s.index(of: element) { return (false, storage.s[i]) }
    return storage.extras.insert(element)
  }
}
```

Of course, there are many problems with this code: for example, it requires Element to be hashable, and it violates Collection requirements by implementing subscript in $O(n \log n)$ time — which is absurdly long. But the problem I find most vexing is that SillySet's indexing subscript modifies its underlying storage by side effect — this breaks my assumptions about what value semantics mean in Swift. (For example, it makes it dangerous to pass SillySet values between threads; even read-only concurrent access may result in data races.)

This particular example might be silly, but the idea of deferring the execution of successive insertions by collecting such elements into a separate buffer has a lot of merit. Creating a sorted set by individually inserting a bunch of elements in a loop isn't very efficient at all. It's a lot quicker to first sort the elements in a separate buffer and then use special bulk loading initializers to convert the buffer into a B-tree in linear time.

Bulk loading works by exploiting the fact that we don't care whether its intermediate states satisfy sorted set requirements — we never look at elements of a half-loaded set.

It's important to recognize opportunities for this type of optimization, because they allow us to *temporarily* escape our usual constraints, often resulting in performance boosts that wouldn't otherwise be possible.

# Farewell

I hope you enjoyed reading this book! I had a lot of fun putting it together. Along the way, I learned a great deal about implementing collections in Swift, and hopefully you picked up a couple of new tricks too.

Throughout this book, we explored a number of different ways to solve the particular problem of building a sorted set, concentrating on benchmarking our solutions in order to find ways to improve their performance.

However, none of our implementations were complete, as the code we wrote was never really good enough for production use. To keep the book *relatively* short and to the point, we took some shortcuts that would be inappropriate to take in a real package.

Indeed, even our `SortedSet` protocol was simplified to the barest minimum: we cut most of the methods of `SetAlgebra`. For instance, we never discussed how to implement the `remove` operation. Somewhat surprisingly, it's usually much harder to remove elements from a balanced tree than it is to insert them. (Try it!)

We didn't spend time examining all the other data types we can build out of balanced search trees. Tree-based sorted maps, lists, and straightforward variants like multisets and multimaps are just as important as sorted sets; it would've been interesting to see how our code could be adapted to implement them.

We also didn't explain how these implementations could be tested. This is a particularly painful omission, because the code we wrote was often tricky, and we sometimes used unsafe constructs, where the slightest mistake could result in scary memory corruption issues and days of frustrating debugging work.

Testing is hugely important; unit tests in particular provide a safety net against regressions and are pretty much a prerequisite to any kind of optimization work. Data structures lend themselves to unit testing especially well: their operations take easy-to-generate input and they produce well-defined, easily validatable output.

Powerful packages like SwiftCheck provide easy-to-use tools for providing full test coverage.

That said, testing COW implementations can be a challenging task. If we aren't careful enough about not making accidental strong references before calling `isKnownUniquelyReferenced`, our code will still produce correct results — it will just do it a lot slower than we expect. We don't normally check for performance issues like this in unit tests, and we need to specially instrument our code to easily catch them.

On the other hand, if we simply forget to ensure we don't mutate shared storage, our code will also affect variables holding unmutated copies. This kind of *unexpected action at a distance* can be extremely hard to track down, because our operation breaks values that aren't explicitly part of its input or output. To ensure we catch this error, we need to write unit tests that specifically check for it — normal input/output checks won't necessarily detect it, even if we otherwise have 100% test coverage.

We did briefly mention that by adding element counts to the nodes of a search tree, we can find the $i$th smallest or largest element in the tree in $O(\log n)$ time. This trick can in fact be generalized: search trees can be augmented to speed up the calculation of any associative binary operation over arbitrary ranges of elements. Augmentation is something of a secret weapon in algorithmic problem solving: it enables us to easily produce efficient solutions to many complicated-looking problems. We didn't have time to explain how to implement augmented trees or how we might solve problems using them.

Still, this seems like a good point to end the book. We've found what seems to be the best data structure for our problem, and we're now ready to start working on building it up and polishing it into a complete, production-ready package. This is by no means a trivial task: we looked at the implementation of half a dozen or so operations, but we need to write, test, and document dozens more!

If you liked this book and you'd like to try your hand at optimizing production-quality collection code, take a look my BTree package on GitHub. At the time of writing, the most recent version of this package doesn't even implement some of the optimizations in our original B-tree code, much less any of the advanced stuff in Chapter 7. There's lots of room for improvement, and your contributions are always welcome.

# How This Book Was Made

This book was generated by *bookie*, my tool for creating books about Swift. (*Bookie* is of course the informal name for a bookmaker, so the name is a perfect fit.)

Bookie is a command-line tool written in Swift that takes Markdown text files as input and produces nicely formatted Xcode Playground, GitHub-flavored Markdown, EPUB, HTML, LaTeX, and PDF files, along with a standalone Swift package containing all the source code. It knows how to generate playgrounds, Markdown, and source code directly; the other formats are generated by pandoc after converting the text into pandoc's own Markdown dialect.

To verify code samples, bookie extracts all Swift code samples into a special Swift package (carefully annotated with `#sourceLocation` directives) and builds the package using the Swift Package Manager. The resulting command-line app is then run to print the return value of all lines of code that are to be evaluated. This output is then split, and each individual result is inserted into printed versions of the book after its corresponding line of code:

```
func factorial(_ n: Int) -> Int {
  return (1 ... max(1, n)).reduce(1, *)
}
▶ factorial(4)
  24
▶ factorial(10)
  3628800
```

(In playgrounds, such output is generated dynamically, but the printout has to be explicitly included in the other formats.)

Syntax coloring is done using SourceKit, like in Xcode. SourceKit uses the official Swift grammar, so contextual keywords are always correctly highlighted:

```
var set = Set<Int>() // "set" is also a keyword for defining property setters
set.insert(42)
▶ set.contains(42)
  true
```

Ebook and print versions of this book are typeset in *Tiempos Text* by the Klim Type Foundry. Code samples are set in Laurenz Brunner's excellent *Akkurat*.

Bookie is not (yet?) free/open source software; you need to contact us directly if you're interested in using it in your own projects.